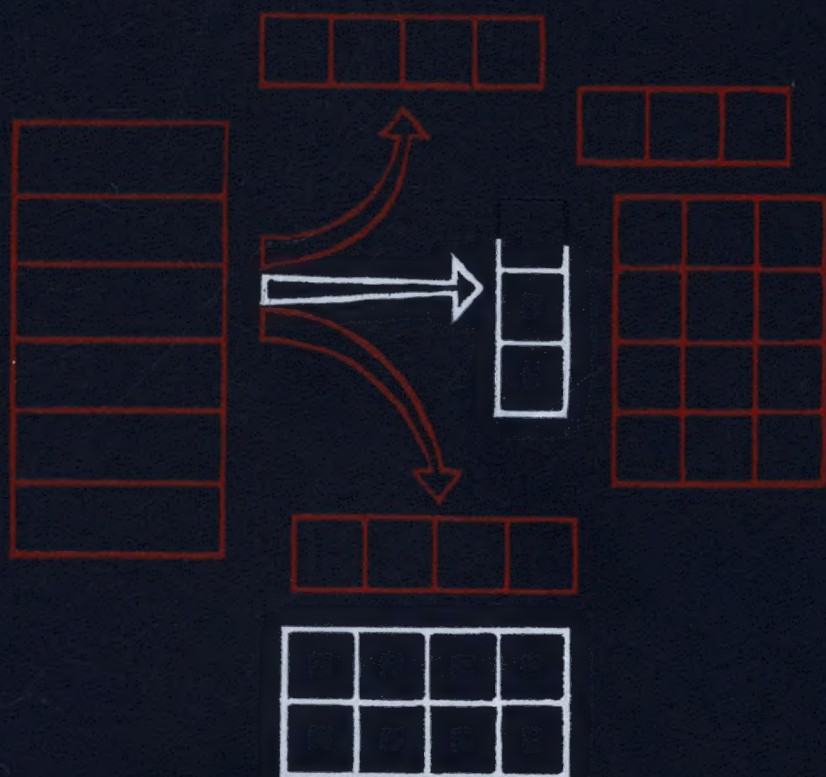


Ю. Василеску

# Прикладное программирование на языке Ада



Издательство „Мир“



```
( FORM_IN_TRANS.V  
AST_TR /= 1  
n      -- There are  
      -- for this d  
      -- transaction  
READ( TRANS_FILE, LO  
e next record  
R.TR_LINE.TRANS_NEXT  
ITIVE(TRANS_IO.SIZE
```











# Ada Programming with Applications

**Eugen N. Vasilescu**

Hofstra University

**Allyn and Bacon, Inc.**

**Boston London Sydney Toronto**



**Ю. Василеску**  
**Прикладное**  
**программирование**  
**на языке Ада**

Перевод с английского

**А. А. Титова**

под редакцией

канд. техн. наук **В. Н. Соболева**



**Москва «Мир» 1990**

ББК 22.18  
В19  
УДК 519.682

**Василеску Ю.**

**В19 Прикладное программирование на языке Ада: Пер. с англ. – М.: Мир, 1990. – 348 с., ил.**

**ISBN 5-03-001108-0**

В книге ученого из США излагаются методы и приемы прикладного программирования на языке Ада, которые могут быть использованы при создании системы управления параллельными процессами. Значительное внимание уделяется алгоритмизации управленческих задач, возникающих в хозяйственной практике. Изложение иллюстрируется множеством примеров, позволяющих читателям легко освоить предлагаемые методы.

Для студентов, изучающих программирование и специалистов в области вычислительной техники.

2404010000-034  
В ————— 134-90 г.  
041 (01)-90

**ББК 22.18**

*Редакция литературы по информатике и робототехнике*

**ISBN 5-03-001108-0 (русск.)**  
**ISBN 0-205-08744-2 (англ.)**

© 1987 by Allyn and Bacon, Inc.  
© перевод на русский язык, А. А. Титов, 1990



## ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Вниманию читателей предлагается книга по программированию на Аде—современном многоцелевом языке, включающем множество полезных особенностей ряда языков-предшественников (от Фортрана и ПЛ/1 до Модулы-2 и Паскаля). Ада в значительной мере базируется на Паскале, однако намного сложнее и мощнее его. В разработке этого языка приняли участие лучшие специалисты разных стран. Язык Ада хорошо приспособлен для программирования как задач вычислительного характера, так и систем реального времени и параллельной обработки. Он удобен в качестве средств системного проектирования, что объясняется такими его особенностями, как наличие пакетов, разделение спецификаций и тел программных модулей, возможность параллельного выполнения нескольких задач, обеспечиваемая механизмом рандеву.

В последнее время язык Ада получил распространение за рубежом как средство разработки сложных программных систем. Трансляторы с Ады имеются и на отечественных ЭВМ, однако широкое применение этого языка в нашей стране сдерживается рядом причин, в том числе отсутствием ориентированных на пользователя изданий, написанных на высоком методическом уровне и содержащих полное и точное изложение современной версии данного языка и методики его применения.

В книге достаточно полно изложены методика и приемы прикладного программирования на Аде. Она является хорошим подспорьем при самостоятельном изучении языка. Изложение опирается на большое количество тщательно подобранных примеров, доведенных до законченных программ. К достоинствам книги относится также умелое дозирование материала. Значительное внимание уделяется применению языка для алгоритмизации управленческих задач, возникающих в хозяйственной практике. Рассмотрены вопросы структурного проектирования, складывающегося из разработки больших программ, и структурного программирования, сводящегося к способам реализации программных модулей. Четко отражена концепция пакетов, являющаяся мощным инструментом создания прикладных программ и позволяющая выделять структуры данных, отделять интерфейс от практической реализации, изолировать объекты и связанные с ними операции. Описанный аппарат весьма удобен при разработке СУБД и создании прикладного обеспечения для управления параллельными процессами.

Книга написана доступным языком, отличается ясностью и четкостью изложения. Охват материала весьма широк. Надеемся, что книга окажет существенную помощь системным аналитикам, прикладным программистам и специалистам по информационным системам. Она может служить учебным пособием для аспирантов и студентов высших учебных заведений.

*В. Н. Соколов*

# ПРЕДИСЛОВИЕ

Ада — язык программирования, разработанный по заказу министерства обороны США с целью противостояния нарастающему кризису в области разработки программных средств. Ада создавалась как язык, предназначенный для встроенных вычислительных систем. В Аду включены: параллельная обработка, работа в реальном масштабе времени, обработка исключительных ситуаций и развитые средства ввода-вывода. Однако эти особенности языка делают его пригодным и для более широкой области применения. Понятие пакета оказалось весьма полезным инструментом при разработке программ. Оно позволяет вводить абстрактные структуры данных и отделять спецификацию сегмента программы (т.е. интерфейс) от его тела (т.е. от фактической реализации), а также дает возможность «скрывать» от пользователя объекты и связанные с ними операции. Наличие задач в языке Ада играет важную роль при разработке систем управления базами данных, а также в тех сферах приложения языка, где необходимо управление параллельными процессами.

По мнению ряда ученых, Ада имеет хорошие шансы стать ведущим языком программирования 80-х гг. В связи с этим, было опубликовано несколько книг по Аде, ориентированных в первую очередь на специалистов по вычислительной технике. Это создало условия для перехода к Аде от таких языков программирования, как Фортран, ПЛ/1 и Паскаль.

Данная книга, однако, в большей степени ориентирована на экономические приложения Ады и, в частности, на создание информационных систем, построенных на базе ЭВМ. Как часто отмечается, основной целью курса по информационным системам является подготовка выпускника высшего учебного заведения для работы в качестве начинающего системного аналитика, прикладного программиста или специалиста по информационным системам. Данная книга соответствует этой цели. Ею могут воспользоваться студенты-дипломники или студенты старших курсов, а также специалисты по информационным системам. Она может быть использована для чтения одно- или двухсеместрового курса. При этом предполагается, что читатель уже имеет по крайней мере начальные знания по ЭВМ и обладает некоторыми элементарными познаниями по одному из языков программирования типа Бейсик, Кобол, Паскаль, Фортран или ПЛ/1.

Книга является самостоятельным учебным пособием. Она преследует следующие цели:

- привить понимание языка программирования со строгой типизацией и разъяснить его преимущества для эффективной и надежной реализации пакетов;
- дать практические знания и опыт в разработке пакетов для экономических задач;



— дать общее понимание таких сложных понятий, как локализация данных, параллельность и модульность.

Читатели не перегружаются материалом. Синтаксис Ады вводится постепенно, при этом делается упор на представление целостных программ на Аде.

К концу первой главы читатель получает сведения о типах и операторах, достаточные для написания простых программ. Гл. 2 и 3 знакомят с такими типами, как действительные, регулярные, комбинированные и ссылочные и с такими операторами, как цикл `for`. В гл. 4 вводятся другие операторы Ады (например, оператор выбора `case`) и такие понятия, как операции участия, комбинированные типы с вариантами. В гл. 5 и 6 излагаются сведения о подпрограммах и детально рассматриваются преобразования типов, инструкции транслятору, области действия идентификатора и правила видимости. Таким образом, в первых шести главах книги обсуждаются понятия, которые можно найти и в других языках.

Во второй части книги (последние пять глав) рассматриваются концепции, введенные в Аду, благодаря которым этот язык может успешно применяться для программирования встроенных вычислительных систем и для разработки сложных программных систем вообще. В этой части описываются: пакеты, развитые средства ввода-вывода Ады, раздельная компиляция, параллельная обработка, обработка исключительных ситуаций.

Ада — сложный язык, а данная книга не рассчитана на использование в качестве справочника по этому языку. В ней не затрагиваются некоторые возможности Ады, наличие которых не очень существенно для большинства читателей. Например, в книге приведены сведения об использовании родовых средств, и в то же время опущены подробности правил написания родовых подпрограмм и пакетов. Хотя родовые средства важны и ценны при разработке программ для управления производством, они не столь необходимы в книге, ориентированной на пользователей Ады, занимающихся информационными системами. В книге не приведено описание средств Ады, зависящих от конкретной реализации языка. Знание особенностей конкретной версии Ады необходимо системному программисту, но далеко не столь важно для лиц, занимающихся прикладными экономическими задачами. Однако знание этих особенностей потребуется при использовании в программах на Аде внешних файлов, созданных с помощью иных языков программирования, а также при желании воспользоваться библиотеками программ, написанными на других языках.

Содержание книги базируется на конспектах, подготовленных автором для чтения курса лекций по программированию на языке Ада для студентов старших курсов Университета Хофстра, специализирующихся по информационным системам в сфере экономики. Программы, приведенные в книге, подвергались трансляции и в значительной степени свободны от синтаксических и семантических ошибок. Ранние версии программ транслировались компилятором JANUS/ADA, а более поздние — с помощью сертифицированных трансляторов Data General/ROLM и VAX-11 компании Digital Equipment Corporation. Для желающих предлагается дискетта, содержащая последние версии программ из данной книги и примеры протоколов трансляции и выполнения программ. Заявки на поставку этой дискетты следует адресовать непосредственно автору книги.

**Выражение признательности**

Автор хотел бы поблагодарить многих лиц, внесших ценный вклад при подготовке настоящей книги. В особенности автор приносит благодарность М. Берковицу за его тщательную, глубокую и хорошо сбалансированную критическую оценку рукописи книги. Автор выражает признательность С. Денбауму за большое число прекрасных и детальных замечаний, способствовавших улучшению как стиля, так и содержания рукописи. Автор благодарит рецензентов книги, Дж. Блейсделла и Ж. Мотиля, за их весьма полезные замечания. Автор приносит благодарность редактору книги, Дж. Сульжицки, за его поддержку и профессиональное руководство. Но прежде всего автор выражает признательность своей жене, Симоне, за ее терпение и понимание, проявленные во время подготовки данной книги.



# Глава 1

## Введение

### 1.1. ЗНАКОМСТВО С ЯЗЫКОМ АДА

#### 1.1.1. Набор символов

В программе на языке Ада используются буквы верхнего регистра, цифры и следующие специальные символы:

" # ' ( ) \* + , - . / : ; < = > \_ |

а также символ пробела. Все вместе они образуют *основной набор символов* языка Ада. Любая программа на Аде может быть написана с помощью только этих символов.

В некоторых реализациях Ады применяется *расширенный набор символов*. В дополнение к основному набору, в расширенном наборе используются буквы нижнего регистра и такие специальные символы:

! \$ % ? @ [ \ ] ^ \_ { } ~

Если сложить 52 (число букв обоих регистров), 10 (количество цифр) и 33 (общее число специальных символов), то получится в итоге 95. Эти 95 символов и составляют 95-символьный набор графических знаков кода ASCII.

В данной книге используется расширенный набор символов. В приложении В приведен полный список всех 128 символов кода ASCII, который включает в себя и 95-символьный набор знаков языка Ада.

#### 1.1.2. Пример простой программы на Аде

Читатель, обладающий даже элементарными знаниями по одному из языков программирования, без большого труда сможет понять приведенную ниже программу MAX3. Она считывает три целых числа и выводит на печать наибольшее среди них.

##### Программа MAX3

```
-- Это комментарий. В Аде комментарии начинаются
-- с двух тире и продолжаются до конца строки.
-- Первые 3 строки этой программы - комментарии.
with TEXT_IO; use TEXT_IO; -- Пакет TEXT_IO
-- делается доступным для программы MAX3. Теперь
-- можно использовать подпрограммы, включенные в
-- состав этого пакета, например NEW_LINE, GET,
-- PUT. Подробные сведения о пакетах и подпрограм-
-- мах приведены в разд. 1.5.
procedure MAX3 is -- Имя процедуры - MAX3,
-- оно задается здесь.
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
-- Предыдущие две строки сделали возможным ис-
-- пользование подпрограмм ввода-вывода GET и
-- PUT для целых чисел. INTEGER_IO является
```

```

-- частью пакета TEXT_IO (см. разд. 1.5).
I, J, K, L : INTEGER;
-- Четыре переменные I, J, K, L объявлены как
-- переменные целого типа.
begin
    -- Объявления завершены, дальше рас-
    -- полагаются операторы.
    GET(I); GET(J); GET(K);
    -- Используется подпрограмма GET из пакета
    -- INT_IO. Три предыдущих оператора выполняют
    -- считывание трех целых чисел, которые будут
    -- размещены в оперативной памяти ЭВМ по адре-
    -- сам, соответствующим переменным I, J и K.
    if I > J
    then
        L := I;    -- Наибольшее значение среди I и J
    else
        L := J;    -- помещается по адресу, соответ-
    end if;        -- ствующему L.
    if L < K
    then
        L := K;    -- Целое число, размещающееся в L,
    end if;        -- заменяется целым значением K
    -- только в том случае, когда L < K.
    -- Теперь наибольшее целое значение
    -- присвоено L.
    NEW_LINE;     -- Используется подпрограмма
    -- NEW_LINE из пакета TEXT_IO. Она
    -- применяется только для выходных
    -- файлов и выполняет переход к но-
    -- вой строке.
    PUT(" The largest is : ");
    -- На экране дисплея или на бумаге появляется
    -- сообщение The largest is :
    PUT ( L );
    -- Вслед за предыдущим текстом будет выведено
    -- значение целого числа, расположенного по ад-
    -- ресу, связанному с L.
    NEW_LINE;
end MAX3;
-- Здесь процедура заканчивается.

```

Это целостная главная программа на Аде. Она состоит из процедуры MAX3 и некоторой дополнительной информации о контексте, необходимой для трансляции и выполнения. Контекст описан строкой:

```
with TEXT_IO; use TEXT_IO;
```

которая открывает доступ данной программе к средствам текстового ввода-вывода. Эти средства необходимы для считывания (с терминала или иного устройства) и записи (на терминал, принтер и т. п.) символов из расширенного набора в таком виде, который может воспринимать человек.

Сама процедура начинается строкой

```
procedure MAX3 is
```

в которой задается имя процедуры и которая помечает начало тела процедуры. Процедура заканчивается строкой

```
end MAX3;
```

Каждая строка программы на языке Ада состоит из последовательности лексических единиц. *Лексические единицы* в главной программе MAX3 — это идентификаторы,

разделители и строки (если не принимать во внимание комментариев). Эти термины будут более подробно освещены в разд. 1.2. Например, в программе MAX3 идентификаторами являются

```
with MAX3 I J else
```

разделителями являются:

```
, ; ()
```

а строка — это

```
"The largest is :"
```

В программах на Аде встречаются и два других вида лексических единиц: числовые литералы и символьные литералы. Числовые литералы представляют собой целые или действительные значения, например:

```
12.0 8.5e1 144
```

Символьным литералом может быть любой из 95 символов расширенного набора, заключенный в апострофы, например:

```
'a' '5'
```

Тело процедуры MAX3 имеет две части. Первая из них — *декларативная часть*. Она расположена между идентификаторами `is` и `begin`. Вторая — *исполняемая часть*. Это последовательность операторов, размещенная между идентификаторами `begin` и `end`.

В декларативной части объявляются и делаются доступными все логические ресурсы, которые должны быть использованы в процедуре. Пусть, например, нужно открыть доступ к процедурам `GET` и `PUT`, которые выполняют ввод или вывод целых чисел в исполняемой части программы MAX3. Это осуществляется такими двумя строками в декларативной части процедуры MAX3:

```
package INT_IO is new INTEGER_IO (INTEGER);
use INT_IO;
```

С помощью этих строк из пакета `INTEGER_IO`, являющегося частью пакета `TEXT_IO`, создается нужный пакет `INT_IO`. Более подробные сведения будут даны в разд. 1.5.1. Пакет `INT_IO` обеспечивает доступ к нескольким версиям процедур `GET` и `PUT`, производящим чтение и запись целых чисел. Выбор конкретной версии этих процедур будет зависеть от формата данных и типа периферийного устройства.

Строкой программы

```
I, J, K, L : INTEGER;
```

вводятся четыре логических объекта целого типа. Если потребуются дополнительные объекты, то перед идентификатором `begin` следует добавить соответствующие объявления.

После запуска программы MAX3 ее исполняемая часть будет выполняться, начиная с трех обращений к процедуре `GET`. При этом три целых числа будут считаны и размещены в адресах памяти, соответствующих переменным `I`, `J` и `K`. Целые числа вводятся в свободном формате, т.е. их позиция в строке и пробелы, окружающие числа, во внимание не принимаются. Затем выполняются операторы `if`. Каждый оператор `if` (если) начинается с идентификатора `if` и заканчивается идентификаторами `end if`, за которыми следует разделитель `;;`. Операторы `if` являются *составными операторами*, поскольку они могут содержать в своем составе и другие операторы. В приведенном примере они содержат *операторы присваивания*. Один из них имеет вид

```
L := I;
```

Его действие заключается в том, что значение переменной `I` размещается по адресу, соответствующему `L`. Как можно догадаться, операторы присваивания и обращения к

процедурам — это *простые операторы*, поскольку они не содержат других операторов. Остальные операторы программы MAX3 являются обращениями к процедурам из пакетов TEXT\_IO и INT\_IO.

Каждый оператор языка Ада заканчивается символом ";". На одной строке может располагаться несколько операторов. Обычно операторы выполняются последовательно. Но такое выполнение может быть прервано некоторыми операторами (например, return) или из-за возникновения ошибок, или особых событий. Ада реагирует на эти ошибки или особые события с помощью *возбуждения исключительных ситуаций*. Подробные сведения об исключительных ситуациях приводятся в гл. 11, а некоторые вводные пояснения даются в разд. 1.5.3.

### 1.1.3. Циклы while в языке Ада

Далее приводится пример еще одной законченной программы на Аде с именем MAXALL. В ней используется цикл while (до\_тех\_пор\_пока). Операторы программы размещены более плотно, чем в предыдущем примере. Программа печатает наибольшее из вводимых целых чисел. Признаком конца входного потока положительных целых чисел считается число -1.

#### Программа MAXALL

```
with TEXT_IO; use TEXT_IO;
procedure MAXALL is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  I, MAX_NO : INTEGER;
begin
  MAX_NO := 0; GET(I);
  while I /= -1 loop
    if I > MAX_NO then MAX_NO := I; end if;
    GET(I);
  end loop;
  PUT(" The largest is : "); PUT(MAX_NO);
  NEW_LINE;
end MAXALL;
```

В программе MAXALL операторы, расположенные между идентификатором loop (цикл) и идентификаторами end loop (конец цикла), выполняются до тех пор, пока выражение, стоящее после идентификатора while, истинно. В данном случае это происходит до тех пор, пока I не станет равным -1. Последовательность символов /= (косая черта и знак равенства) означает «не равно». Это — составной разделитель. Здесь опять-таки целые числа вводятся в свободном формате, возможно, по несколько чисел в строке.

Читатель может выразить недоумение по поводу того, что для записи и считывания столь простых объектов, какими являются целые числа, потребовался такой тщательно разработанный процесс. Но пока лучше не спешить с выводами и подождать до тех пор, пока станет возможным более детальное обсуждение концепции пакетов. До этого момента рекомендуется изучать примеры, приведенные автором, не задавая себе вопросы о необходимости использования пакетов. Если имена файлов не указаны явно, то в большинстве диалоговых систем подразумевается, что входные данные поступают в систему с клавиатуры, а выходные данные посылаются на терминал. При работе в диалоговой системе для ввода программы следует пользоваться редактором текстов. Для запуска программы нужно применять имеющиеся в вашей системе команды, выполняющие вызов транслятора с Ады и редактора связей.

## 1.2. ДАЛЬНЕЙШИЕ СВЕДЕНИЯ О ЛЕКСИЧЕСКИХ ЕДИНИЦАХ

Любую программу, написанную на языке Ада, можно рассматривать как последовательность лексических единиц. Несколько лексических единиц могут располагаться в одной строке, но в то же время лексическая единица не должна занимать более чем одну строку. Как уже отмечалось, лексические единицы — это идентификаторы, числовые и символьные литералы, строки, разделители и комментарии. Идентификаторы и числовые литералы должны отделяться от других идентификаторов или числовых литералов по крайней мере одним пробелом или должны располагаться на разных строках.

*Идентификаторы* в Аде — это последовательности букв, цифр и символов подчеркивания. При этом первым символом последовательности должна быть буква, а по бокам каждого символа подчеркивания должны стоять буквы или цифры. Буквы верхнего и нижнего регистров, входящие в состав идентификаторов, не различаются. Поэтому два идентификатора будут считаться одинаковыми, если они отличаются друг от друга только видом регистра для входящих в них букв, а в остальном последовательности их символов совпадают.

**Пример.** Ниже приведены правильные идентификаторы языка Ада:

```
I Q12_pqR   General_Ledger   ACCOUNT_123
ZZZZ   Check_CREDIT_APPROVAL   INTEREST
```

А вот примеры неправильных идентификаторов:

1Expect	Последовательность символов начинается с цифры
MORENO_	Символ подчеркивания не окружен буквами или цифрами
TOO_ _MANY	То же

Некоторые из идентификаторов являются *зарезервированными словами*, имеющими особое значение в языке Ада. Их нельзя использовать в качестве имен иных ресурсов. В Аде насчитывается 62 зарезервированных слова. Их список приведен в табл. 1.1.

Таблица 1.1. Зарезервированные слова Ады

abort	declare	generic	of	select
accept	delay	goto	or	separate
access	delta		others	subtype
all	digits	if	out	
and	do	in		task
array		is	package	terminate
at	else		pragma	then
	elsif	limited	private	type
	end	loop	procedure	
begin	entry		raise	use
body	exception	mod	range	
	exit		record	when
			rem	while
case	for	new	renames	with
constant	function	not	return	
		null	reverse	xor



Как указывалось выше, числовые литералы представляют собой целые и действительные числа. Для улучшения читабельности литералов среди цифр можно поместить и символы подчеркивания. Действительный литерал содержит десятичную точку, а в целом литерале она отсутствует. Целые литералы могут содержать мантиссу с положительным показателем степени. Примеры числовых литералов:

Целые литералы 0 5 13 21e + 3 21\_345 34 21e + 2  
 Действительные литералы 1.3 2.0e - 12 1\_2\_3\_45.08E; + 7 1\_123.456

Числовые литералы можно записывать в любой системе счисления с основанием от 2 до 16. Если основание не равно 10, то перед числовым литералом ставится основание системы счисления (в десятичном виде) и символ "#". Например, 2#1110 представляет десятичное целое число 14, записанное в двоичной системе счисления. Если при этом имеется показатель степени, то он записывается в десятичной системе. Например, десятичное число 56 можно записать в двоичном виде как 2#111e + 3. Если основание системы счисления превышает 10, то применяются латинские буквы от A до F, соответствующие цифрам от 10 до 15.

Как уже упоминалось, символьный литерал — это любой из 95 символов графического набора кода ASCII, заключенный в апострофы. Например, символьными литералами являются

'a' 'A' '2' ' ' '5'

Не следует путать символьные литералы со строками. Строки получаются, если несколько символов (ноль и более) заключить в кавычки. Пример строк:

"a" "Just a regular string" "1.234;" "2.1 cents"

*Пустая строка* (в ней нет символов) представляется в виде "". Если необходимо поместить в саму строку символ ", то его следует дублировать, т.е. записывать в виде "". Длинные строки, занимающие более одной строки текста программы, можно представить в виде совокупности более мелких строк, при этом используется символ сцепления, обозначаемый &. Например:

"if you need a really long string like a heading " &  
 "you can use the symbol ""&"" to catenate—i.e.," &  
 "glue together—the pieces from several lines"

Обратите внимание на то, что здесь внутри строки использованы кавычки повторяющиеся два раза, т.е. применена конструкция ""&"". При этом будет напечатано "&"

В используемой вами версии языка Ада может оказаться возможным применение знаков, не входящих в основной набор символов. В таком случае любой символ, не входящий в этот набор, будет преобразован в эквивалентное представление, соответствующее основному набору, с помощью идентификаторов пакета, названного ASCII (см. приложение В). Применение пакетов будет описано в гл. 7.

Как упоминалось выше, начало комментария отмечается двумя тире. Комментарий заканчивается концом соответствующей строки текста программы. Перед комментарием могут располагаться операторы языка Ада, однако вслед за ним на той же строке операторов уже быть не может.

## 1.3. СВЕДЕНИЯ О ТИПАХ И ОБЪЕКТАХ

### 1.3.1. Целые и перечисляемые типы и связанные с ними объекты

За небольшими исключениями каждый идентификатор (если он не является ключевым словом Ады) должен быть описан в явном виде, и его свойства должны быть известны перед тем, как он может быть использован в исполняемой части программы. В основном идентификаторы описываются с помощью объявлений, появляющихся в

декларативной части программы. Грубо говоря, в этом случае *объявление* дает сведения о том, каким видом ресурса является описываемый идентификатор. Например, в предыдущих двух программах были объявлены переменные I, J и K. Эти описания были необходимы для того, чтобы оперировать с целыми числами при помощи приведенных идентификаторов. Другим видом ресурсов, который может быть обозначен идентификатором, является тип. *Типы* играют главенствующую роль в Аде. Они представляют собой некоторые шаблоны или модели данных.

Общая форма объявления типа такова:

type имя\_типа is определение\_типа;

(Подчеркнуты зарезервированные слова языка Ада.) Имя\_типа — это идентификатор, выбранный программистом.

Типы используются для определения совокупностей значений и операций, разрешенных для них. В этом подразделе рассматриваются два важных вида типов — целые типы и перечисляемые типы, называемые *дискретными типами*.

Для *целых типов* член определение\_типа в объявлении типа — это *уточнение диапазона значений*, состоящее из зарезервированного слова *range*, после которого следуют нижняя граница диапазона, две точки и затем верхняя граница. Нижняя граница не может быть больше верхней. Вот некоторые примеры объявлений целых типов:

```
type AGE is range 0 .. 200 ;
type DAY_OF_MONTH is range 1 .. 31 ;
type ACCOUNT_NO is range 0 .. 9999 ;
type TEMPERATURE is range -140 .. 2100 ;
type FEVER is range 95 .. 109 ;
```

Для целых типов множество допустимых значений задается в уточнении диапазона значений. Например, значения величин целого типа AGE могут находиться в пределах от 0 до 200 включительно. Обычно для целых типов определены следующие операции: присваивание, сравнение, сложение (+), вычитание (—), умножение (\*), деление (/) и возведение в степень (\*\*). Пять последних операций называются арифметическими.

Для *перечисляемых типов* член «определение\_типа» в объявлении состоит из заключенной в скобки последовательности перечисляемых литералов, разделенных запятыми. *Перечисляемыми литералами* могут быть либо идентификаторы, либо символьные литералы. В следующих примерах представлены объявления перечисляемых типов:

```
type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN) ;
type ACCOUNT is (MARGIN,HEDGE) ;
type SECURITY is (DISCOUNT,COUPON) ;
type SWITCH is (ON,OFF) ;
```

Наборы возможных значений для перечисляемых типов задаются в явном виде с помощью списка этих значений в объявлении типа. Например, для величин, принадлежащих к перечисляемому типу SECURITY, допустимы два значения — перечисляемый литерал DISCOUNT и перечисляемый литерал COUPON.

Среди операций, определенных для перечисляемых типов, следует упомянуть операцию "<" (меньше). Значение одной величины перечисляемого типа будет считаться меньшим, чем значение другой величины того же типа, если в списке перечисляемых литералов для этого типа первое значение появляется раньше, чем второе. Операция ">" определяется сходным образом.

В языке Ада имеется небольшое количество предопределенных типов. *Предопределенные типы* не требуется объявлять в декларативной части подпрограммы. Вот некоторые из них: целый тип INTEGER (целый), перечисляемый тип CHARACTER

(символьный), перечисляемый тип **BOOLEAN** (логический) и тип **STRING** (строковый). (Тип **STRING** принадлежит к разряду типов, описываемых в гл. 2.) Далее дается краткое описание значений и операций для этих предопределенных типов.

Диапазон значений, которые могут принимать величины типа **INTEGER**, зависит от реализации. Поэтому каждый разработчик транслятора с языка Ада самостоятельно принимает решение о том, каким должен быть этот диапазон. Как правило, это по меньшей мере 64К (быть может, от  $-32К$  до  $+32К - 1$ ). Операции, разрешенные для этого предопределенного типа, такие же, как и для любого другого целого типа.

Величины типа **CHARACTER** имеют в качестве значений символы из набора **ASCII**. Операции, допустимые для этого типа, включают присваивание и сравнение. Эти же операции допустимы и для любого перечисляемого типа.

Величины типа **BOOLEAN** могут принимать значения **FALSE** и **TRUE**.

Значениями величины типа **STRING** могут быть последовательности символов, заключенные в кавычки. К величинам этого типа применимы операции сравнения с другими строками, а также операции присваивания и сцепления (как с другими строками, так и с отдельными символами).

После того как тип объявлен, можно объявлять величины этого типа, называемые *объектами*. В языке Ада имеются два вида объектов — константы и переменные. После объявления *переменной* некоторого типа, она может иметь любое значение, допустимое для этого типа. Значение переменной можно изменить, например, путем присваивания ей нового значения. Значения *констант* некоторого типа задаются при их объявлении. Они не могут изменяться впоследствии. Например, в программе **MAX3** переменные **I**, **J**, **K** и **L** объявлены как переменные предопределенного типа **INTEGER**. В программе эти переменные сравниваются и используются в операторах присваивания. Как говорилось выше, эти операции разрешены для переменных типа **INTEGER**.

Приведем примеры объявлений переменных:

```
AGE_EMPL_1 : AGE ;
MORE_AGE, EVEN_MORE_AGE : AGE ;
TRANS_DAY : DAY_OF_MONTH ;
SETTLEMENT_DATE, NOTICE_DAY : DAY_OF_MONTH ;
EMPL_ACC, EMPL_ACC_TEMP : ACCOUNT_NO ;
PATIENT_1_TEMP : FEVER ;
BOILING_TEMP : TEMPERATURE ;
DAY_1, DAY_2 : DAY ;
```

Отметим еще раз, что объявления типов следует располагать раньше, чем объявления объектов, принадлежащих к этим типам. Предопределенные типы объявлять не надо.

Примеры объявления констант:

```
MINIMUM_AGE : constant AGE := 18 ;
DRINKING_AGE : constant AGE := 21 ;
MONTHLY_STATEMENTS_DAY : constant DAY_NO := 1 ;
NORMAL_TEMP : constant FEVER := 98 ;
PAY_DAY : constant DAY := FRI ;
```

Объявления объектов (переменных и констант) в этих примерах составлены по определенному шаблону. Для переменных вид этого шаблона таков:

список\_идентификаторов : имя\_типа;

Для констант форма объявления такая:

список\_идентификаторов : constant имя\_типа := числовой\_литерал;

Список\_идентификаторов — это последовательность идентификаторов, разделенных запятыми. Он представляет имена объектов. Имя\_типа — это предопределенный тип или же тип, введенный программистом. Другие формы объявлений, в состав которых входят различные виды уточнений, будут рассмотрены в следующей главе.

У читателей может возникнуть вполне естественный вопрос: в чем же польза от таких концепций, как типы или уточнения диапазонов значений? Среди многих преимуществ, которые предоставляют данные средства, есть и то, что определение типов вводит некоторые ограничения на пределы возможных значений величин и на действия с этими величинами. Эти ограничения встраиваются в объектный код программы компилятором с языка Ада и автоматически проверяется их соблюдение. Тем самым предотвращаются операции с неверными данными.

### 1.3.2. Использование символьных и строковых типов

Далее показано, каким образом можно переделать программу MAX3 для того, чтобы она выбирала из трех символов тот, номер позиции которого в последовательности символов кода ASCII является наибольшим

#### Программа CHAR\_MAX3

```
with TEXT_IO; use TEXT_IO;
-- Пакет TEXT_IO делается доступным для программы
-- CHAR_MAX3. Теперь можно использовать подпро-
-- граммы из CHAR_MAX3, например NEW_LINE, GET,
-- PUT и т.д.
procedure CHAR_MAX3 is
  I, J, K, L: CHARACTER;
  -- Переменные I, J, K и L об'явлены как пере-
  -- менные символьного типа.
begin
  -- Об'явления закончены, далее располагаются
  -- операторы.
  GET(I); GET(J); GET(K);
  -- Используется подпрограмма GET из пакета
  -- TEXT_IO. Считываются три символа, и их зна-
  -- чения присваиваются переменным I, J и K.
  if I > J
  then
    L := I; -- Переменной L присваивается
  else -- значение, наибольшее среди
    L := J; -- I и J.
  end if;
  if L < K
  then -- Значение L становится равным
    L := K; -- значению K только в том слу-
  end if; -- чае, если L < K в соответствии с
  -- последовательностью ASCII.
  -- Теперь наибольшее символьное значение при-
  -- своено L.
  NEW_LINE;
  -- Используется подпрограмма NEW_LINE из
  -- пакета TEXT_IO.
  PUT(" The largest is : ");
  -- Напечатать или отобразить на дисплее со-
  -- общение: The largest is :
  PUT(L);
  -- А затем напечатать символьное значение,
  -- которое присвоено L.
  NEW_LINE;
end CHAR_MAX3;
-- Конец процедуры.
```

В отличие от аналогичной программы для целых чисел здесь нет необходимости в использовании дополнительной части пакета TEXT\_IO для чтения (GET) или записи (PUT) символов. Символы считываются последовательно. После выполнения каждой операции GET номер позиции во входном файле увеличивается на единицу. Поэтому между тремя вводимыми символами не следует помещать пробелы. Так, для определения символа с наибольшим номером по таблице ASCII среди "Z", "A" и "P" следует ввести ZAP.

Смысл оператора присваивания

`L := K;`

одинаков как для целых, так и для символьных типов. Текущее значение переменной, находящейся слева от составного символа `:=`, заменяется на значение, полученное в результате вычисления выражения (в данном случае – это переменная), расположенного справа. Литералы (числовые и символьные) и переменные – это простые виды выражений, называемые *простейшими* выражениями. Они могут служить основой для построения более сложных выражений. Ада – весьма строгий язык, в котором, как правило, не допускается использовать величины разных типов в одном выражении или в одном операторе присваивания.

**Пример.** Пусть имеются объявления

`M, N : INTEGER;`  
`X, Y : CHARACTER;`

Тогда можно написать следующие операторы присваивания:

`M := M + N;`  
`M := N * 3;`  
`X := 'c';`  
`X := Y;`

Но приведенные ниже операторы присваивания неверны, поскольку в них входят величины разных типов:

`M := M + X;`  
`N := 'A';`  
`X := 5;`

Предопределенный тип STRING будет подробно рассмотрен в следующей главе, которая посвящена изучению регулярных типов. В данной главе, однако, нам требуется объявить переменные типа STRING и использовать их в процедурах GET и PUT из пакета TEXT\_IO. Например:

`F_NAME : STRING (1..25);`

это переменная типа STRING, вмещающая 25 символов. Первый символ находится в 1-й позиции строки, а последний – в 25-й. Длина строки равна 25. Другой пример переменной типа STRING:

`STREET_ADDRESS : STRING (1..20);`

При выполнении процедур GET или PUT с аргументами типа STRING символы строки считываются или записываются, начиная с первой доступной позиции файла. Количество введенных или выведенных символов равно длине строки, являющейся параметром используемой процедуры. Например, оператор

`GET (F_NAME);`

введет следующие 25 символов. Если последняя введенная колонка была, скажем, 13-я, то будут прочитаны символы, начиная с позиций 14-й до 25-й включительно. Аналогично оператор

`PUT (STREET_ADDRESS);`



выполнит запись 20 символов переменной STREET\_ADDRESS, начиная с первой доступной позиции.

Другая разновидность процедуры GET или PUT позволяет указывать число цифр вводимого или выводимого числа. Если имеется объявление

```
I : INTEGER;
```

то оператор

```
GET(I, 5);
```

выполняет ввод целого числа из следующих 5 позиций. Если последняя колонка была 8-й, то целое число будет введено с позиций от 9-й по 13-ю включительно.

Приведенная ниже программа, названная HEAVY, иллюстрирует эти положения. Каждая входная запись содержит в первых 20 позициях фамилию человека, а в следующих 5 позициях — его вес в фунтах. Последняя запись файла содержит букву X в первой позиции, а в остальных — пробелы. Программа выводит фамилию человека, вес которого наибольший.

### Программа HEAVY

```
with TEXT_IO; use TEXT_IO;
procedure HEAVY is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  WEIGHT, MAX_WEIGHT : INTEGER;
  H_NAME, MAX_H_NAME : STRING (1 .. 20);
begin
  MAX_WEIGHT := 0; GET(H_NAME);
  -- Считываются первые двадцать символов стро-
  -- ки. Если отсутствует обращение к подпро-
  -- грамме SKIP_LINE, то при следующем вызове
  -- подпрограммы GET чтение начнется с 21-й
  -- позиции.
  while H_NAME /= "X"
  -- Строки вводятся и обрабатываются до тех
  -- пор, пока не встретится строка, в начале
  -- которой располагаются X и 19 пробелов.
  loop
    GET(WEIGHT, 5);
    -- Из следующих пяти позиций строки, т.е
    -- из позиций 21-25, считывается целое
    -- число, которое присваивается переменной
    -- WEIGHT.
    if WEIGHT > MAX_WEIGHT
    then
      MAX_WEIGHT := WEIGHT;
      MAX_H_NAME := H_NAME;
      -- Этот оператор присваивания применим
      -- только для строк одинаковой длины.
    end if;
    SKIP_LINE;
    -- Эта подпрограмма из пакета TEXT_IO ра-
    -- ботает только со входными файлами. Она
    -- выполняет переход к началу следующей
    -- строки.
    GET(H_NAME);
  end loop;
  NEW_LINE;
```

```

    PUT(" The heaviest person is : ");
    PUT(MAX_H_NAME);
    NEW_LINE;
end HEAVY;

```

Если строки, поступающие на вход программы, будут иметь вид

```

JOHNSON K. Mary      00145
J. K. Peterson      00175
Paul Amaretto       00155
X

```

то программа выведет такой результат

J. K. Peterson

### 1.3.3. Атрибуты целых и перечисляемых типов

Типы и объекты в языке Ада могут иметь некоторые предопределенные характеристики, называемые *атрибутами*. Например, целые и перечисляемые типы имеют одинаковый набор атрибутов. Среди этих атрибутов — FIRST и LAST. Эти атрибуты дают соответственно минимальное и максимальное значения, возможные для величин заданного типа. Ниже приводятся примеры атрибутов:

Атрибут	Описание
INTEGER'FIRST	Дает наименьшее целое значение, обеспечиваемое предопределенным типом INTEGER
INTEGER'LAST	Дает наибольшее целое значение, обеспечиваемое предопределенным типом INTEGER
AGE'FIRST	Дает наименьшее значение, допустимое для типа AGE (см. примеры на с. 15–16); здесь оно равно 0
AGE'LAST	Дает 200 (см. с. 15)
ACCOUNT_NO'FIRST	Дает 0 (см. с. 15)
ACCOUNT_NO'LAST	Дает 9999 (см. с. 15)
DAY'FIRST	Дает MON (см. с. 15)
DAY'LAST	Дает SUN (см. с. 15)

Как показывают примеры, для получения значений атрибутов FIRST и LAST необходимо за именем типа поставить апостроф и имя атрибута.

Для дискретных типов имеются еще 4 атрибута: POS, SUCC, PRED и VAL. При запросе этих атрибутов нужно, помимо имени типа и названия атрибута, указать в скобках некоторое значение (или выражение, результатом которого является некоторое значение).

Атрибут POS дает номер позиции, которую занимает значение указанной величины по отношению к минимально возможному для данного типа значению. Вот некоторые примеры:

Атрибут POS	Значение
AGE'POS(2)	2
DAY'POS(TUE)	1
SWITCH'POS(ON)	0

Атрибут SUCC дает следующее по порядку значение для величины заданного типа:

Атрибут SUCC	Значение
INTEGER'SUCC(7)	8
CHARACTER'SUCC('f')	g
AGE'SUCC(2)	3
DAY'SUCC(TUE)	WED
SWITCH'SUCC(ON)	OFF

AGE'SUCC(200)	Возникнет исключительная ситуация, свидетельствующая об ошибке
DAY'SUCC(SUN)	То же

Атрибут PRED дает предшествующее значение для величины заданного типа.

Атрибут PRED	Значение
INTEGER'PRED(8)	7
CHARACTER'PRED('g')	'f'
AGE'PRED(3)	2
DAY'PRED(TUE)	MON
SWITCH'PRED(OFF)	ON
AGE'PRED(0)	Возникнет исключительная ситуация, свидетельствующая об ошибке
DAY'PRED(MON)	То же

Атрибут VAL возвращает значение, позиция которого (в множестве допустимых для данного типа значений) задается положительным числом.

Атрибут VAL	Значение
CHARACTER'VAL(71)	'F'
AGE'VAL(3)	2
DAY'VAL(2)	TUE
SWITCH'VAL(1)	ON
DAY'VAL(8)	Возникнет исключительная ситуация, свидетельствующая об ошибке

В следующем подразделе приведена программа UP\_MONDAY, которая дает примеры практического применения атрибутов для дискретных типов.

### 1.3.4. Использование целых и перечисляемых типов

Следующая программа, названная UP\_MONDAY, связана с поговоркой, бытующей на Уолл-стрите: Up on Monday, down on Tuesday (подъем в понедельник, спад во вторник). Здесь имеется в виду то, что, если деловая активность испытывает подъем в понедельник, она, по всей вероятности, будет подвергаться спаду во вторник. Программа считывает информацию из каждой входной строки. В строке размещаются два слова. Первое слово — английское название дня недели (от понедельника (Monday) до пятницы (Friday)), а второе слово характеризует изменение деловой активности по сравнению с предшествующим днем (up — рост, down — спад, unchanged — без изменений). На вход программы подается заранее неизвестное количество строк. Условимся, что признаком конца данных будет указание в качестве дня недели воскресенья (Sunday). Программа должна проверять, чтобы перед данными о вторнике на вход подавались данные о понедельнике. Дело в том, что праздники, как и дни выборов<sup>1)</sup>, иногда бывают в понедельник, а иногда — во вторник, и надо проверять, чтобы эта последовательность не нарушалась. Никаких других проверок не делается, хотя в реальных программах они наверняка бы потребовались. Предполагается, что на вход программы поступают по меньшей мере две записи.

#### Программа UP\_MONDAY

```
with TEXT_IO; use TEXT_IO;
procedure UP_MONDAY is
  type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
```

<sup>1)</sup> В такие дни, по всей вероятности, деловая жизнь в США замирает. — Прим. перев.

```

type CHANGE is (DOWN, UNCHANGED, UP);
-- Объявление двух перечисляемых типов.
-- Первый тип имеет значения от MON до SUN, а
-- второй - DOWN, UNCHANGED и UP.
package DAY_IO is new ENUMERATION_IO(DAY);
use DAY_IO;
-- Эти две строки нужны для обеспечения ввода
-- и вывода величин типа DAY. Теперь можно
-- пользоваться свободной формой обращения
-- к GET и PUT для типа DAY. Величины типа
-- DAY могут вводиться с помощью букв как
-- верхнего, так и нижнего регистра, и это не
-- повлияет на результат. Все это справедливо
-- для любого перечисляемого типа.
package CHANGE_IO is new ENUMERATION_IO(CHANGE);
use CHANGE_IO;
-- Как только что пояснялось, эти две строки
-- необходимы для обеспечения ввода-вывода
-- (I/O) для величин типа CHANGE.
CURRENT_DAY, PREVIOUS_DAY : DAY;
-- Объявлены две переменные типа DAY.
CURRENT_CHANGE, PREVIOUS_CHANGE : CHANGE;
-- Объявлены две переменные типа CHANGE.
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
TOTAL_UP_MONDAY, TOTAL_UP_MON_AND_DOWN_TUE :
    INTEGER;
-- Объявлены две переменные предопределенного
-- типа INTEGER.
begin
    TOTAL_UP_MONDAY := 0;
    TOTAL_UP_MON_AND_DOWN_TUE := 0;
    GET(PREVIOUS_DAY);
    GET(PREVIOUS_CHANGE);
    -- Первая строка только что была считана, а
    -- значения типов DAY и CHANGE присвоены пере-
    -- менным, идентификаторы которых начинаются
    -- со слова PREVIOUS (предыдущий).
    SKIP_LINE;
    GET(CURRENT_DAY);
    while CURRENT_DAY /= SUN
    loop
        GET(CURRENT_CHANGE);
        if PREVIOUS_DAY = MON and
           PREVIOUS_CHANGE = UP
        -- С использованием атрибутов эти условия
        -- можно было бы записать так:
        -- PREVIOUS_DAY = DAY'FIRST and
        -- PREVIOUS_CHANGE = CHANGE'FIRST
        then
            TOTAL_UP_MONDAY := TOTAL_UP_MONDAY+1;
            if CURRENT_DAY = TUE and
               CURRENT_CHANGE = DOWN

```

```

-- Эквивалентное условие с использова-
-- нием атрибутов имеет вид
-- CURRENT_DAY      =
-- DAY'SUCC(PREVIOUS_DAY)      and
-- CURRENT_CHANGE    =
-- CHANGE'SUCC(PREVIOUS_CHANGE)
-- Еще один пример записи этого усло-
-- вия:
-- DAY'PREV(CURRENT_DAY)      =
-- PREVIOUS_DAY              and
-- CHANGE'PREV(CURRENT_CHANGE) =
-- PREVIOUS_CHANGE
-- then
--     TOTAL_UP_MON_AND_DOWN_TUE :=
--     TOTAL_UP_MON_AND_DOWN_TUE + 1;
-- end if;
end if;
PREVIOUS_DAY := CURRENT_DAY;
PREVIOUS_CHANGE := CURRENT_CHANGE;
SKIP_LINE;
GET(CURRENT_DAY);
and loop;
NEW_LINE;
PUT(" The total of up days on Mondays is : ");
PUT(TOTAL_UP_MONDAY);
NEW_LINE;
PUT(" The total of up on Mondays and down ");
PUT("on Tuesdays is : ");
PUT(TOTAL_UP_MON_AND_DOWN_TUE);
and UP_MONDAY;

```

Обратите внимание на то, что если при выполнении оператора GET не встретится значение требуемого типа, то будет возбуждена исключительная ситуация, свидетельствующая об ошибке.

В программе UP\_MONDAY для построения сложного выражения из двух простых используется логическая операция and (И). Это фактически — логическое (BOOLEAN) выражение, поскольку тип результата этого выражения — логический. Смысл этого выражения совершенно ясен. В разд. 1.4 будут представлены дополнительные сведения о логических выражениях.

В нижеследующей программе иллюстрируется применение целых и перечисляемых типов. Это — учетная программа. Она считывает информацию о товарах, названия которых хранятся в ведомости. В каждой входной строке даются сведения только об одной позиции товара. Формат строки данных имеет вид

Позиции	Данные
1-4	Номер позиции товара
5-24	Описание товара
25-27	Имеющееся количество
28-30	Заказанное количество
31-50	Место расположения склада

Признаком конца данных служит строка с номером позиции, равным 9999.

Для каждой введенной строки наличное количество товара суммируется с количеством заказанного товара. Выдается предупреждающее сообщение о необходимости дополнительного заказа, если суммарное количество имеющегося и заказанного товара по данной позиции менее 10. После окончания ввода всех строк печатается



сообщение, содержащее номер позиции и описание для товара, количество которого максимально (затоваривание), и место расположения склада с этим видом товара. В конце печатается общее число позиций товара, подлежащего дополнительному заказу.

### Программа INVENTORY

```
with TEXT_IO; use TEXT_IO;
procedure INVENTORY is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type QUANTITY is range 0 .. 999;
  type ITEM_NO is range 0 .. 9999;
  TOTAL_QUANTITY_PER_ITEM,
  TOTAL_REORDERED_ITEMS : QUANTITY;
  HIGH_QUANTITY_PER_ITEM : QUANTITY;
  package QUANT_IO is new INTEGER_IO(QUANTITY);
  use QUANT_IO;
  package ITEM_IO is new INTEGER_IO(ITEM_NO);
  use ITEM_IO;
  QUANT_ON_HAND, QUANT_ON_ORDER : QUANTITY;
  ITEM_NO_IN, HIGH_ITEM_NO_IN : ITEM_NO;
  REORDER_POINT : constant QUANTITY := 10;
  ITEM_DESCRIPTION, HIGH_ITEM_DESCRIPTION :
    STRING(1 .. 20);
  type WAREHOUSE is
    (ILLINOIS, NEW_YORK, TEXAS, CALIFORNIA, FLORIDA);
  HIGH_WAREHOUSE, WAREHOUSE_IN : WAREHOUSE;
  package WAREHOUSE_IO is new
    ENUMERATION_IO(WAREHOUSE);
  use WAREHOUSE_IO;
begin
  TOTAL_REORDERED_ITEMS := 0;
  HIGH_QUANTITY_PER_ITEM := 0;
  GET( ITEM_NO_IN, 4 );
  -- Здесь используется пакет ITEM_IO.
  while ITEM_NO_IN /= 9999
  loop
    GET(ITEM_DESCRIPTION);
    GET(QUANT_ON_HAND, 3);
    -- Здесь необходим пакет QUANT_IO.
    GET(QUANT_ON_ORDER, 3);
    GET(WAREHOUSE_IN);
    TOTAL_QUANTITY_PER_ITEM := QUANT_ON_HAND +
                                QUANT_ON_ORDER;
    if HIGH_QUANTITY_PER_ITEM <
      TOTAL_QUANTITY_PER_ITEM
    then
      HIGH_QUANTITY_PER_ITEM :=
        TOTAL_QUANTITY_PER_ITEM;
      HIGH_ITEM_NO_IN := ITEM_NO_IN;
      HIGH_ITEM_DESCRIPTION := ITEM_DESCRIPTION;
      HIGH_WAREHOUSE := WAREHOUSE_IN;
    end if;
    if TOTAL_QUANTITY_PER_ITEM < REORDER_POINT;
    then
      NEW_LINE;
      PUT(ITEM_DESCRIPTION);
```

```

PUT(
  " *** This item has to be reordered *** ");
NEW_LINE;
TOTAL_REORDERED_ITEMS :=
  TOTAL_REORDERED_ITEMS + 1;
end if;
SKIP_LINE;
GET(ITEM_NO_IN);
end loop;
NEW_LINE;
PUT(" The highest inventory level is for ");
PUT(HIGH_ITEM_NO_IN);
PUT(HIGH_ITEM_DESCRIPTION);
NEW_LINE;
PUT(" Located in ");
PUT(HIGH_WAREHOUSE);
NEW_LINE;
PUT(" The total number of items needed to be " &
  " reordered is ");
PUT( TOTAL_REORDERED_ITEMS, 4 );
end INVENTORY;

```

## 1.4. СВЕДЕНИЯ О ВЫРАЖЕНИЯХ

До сих пор нам встречались выражения, располагавшиеся справа от составного символа ":"=" в операторах присваивания, а также после зарезервированного слова if (если) в операторах if. Эти выражения были весьма простыми и состояли в основном из простейших выражений, которыми в наших примерах являлись литералы, константы и переменные.

*Выражения* представляют собой формулы, по которым вычисляются значения. Для литералов, констант и переменных (т.е. для только что упомянутых простейших выражений) значение, вычисляемое по формулам,—это значение, связываемое непосредственно с данным литералом или идентификатором. Однако простейшие выражения могут комбинироваться с помощью операций и образовывать более сложные выражения.

*Операции* применяются к операндам. Если для операции требуется только один операнд, то она называется *унарной* (*одноместной*). Например, not—это унарная логическая операция. Символы "+" и "-" при употреблении их как знаков числа (положительный и отрицательный) обозначают унарные арифметические операции. Если для применения операции требуются два операнда, то она называется *бинарной операцией*. Например, "+" при сложении и "\*" при умножении—это арифметические бинарные операции. Зарезервированное слово and представляет логическую бинарную операцию.

Для операций установлены приоритеты, определяющие порядок их выполнения. Операции с более высоким приоритетом выполняются раньше, чем операции с низким приоритетом. Если несколько операций имеют одинаковый приоритет, то они выполняются слева направо. Порядок выполнения операций может быть изменен с помощью скобок. В этом случае выражения, стоящие в скобках, вычисляются первыми. Ниже приведен полный список предопределенных операций, которые применимы к целым типам. Операции перечислены в порядке убывания приоритетов, на одной строке даны операции с одинаковыми приоритетами.

**	abs	Возведение в степень, вычисление абсолютной величины числа
*	/, mod, rem	Умножение, деление, остаток от деления (сохраняется знак делителя), остаток от деления (сохраняется знак делимого)

+, -	Знаки (унарные операции)
+, -	Сложение, вычитание (бинарные операции)
=, /=, <, <=, >, >=, in, not in	

В последней строке приведены операции с наименьшим приоритетом, применяемые к целым числам. Смысл этих операций такой:

=	Равенство
/=	Неравенство
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно
in	Проверка принадлежности
not in	То же

Операции in и not in называются *операциями проверки принадлежности*, а остальные операции — *операциями отношений*. Для перечисляемых типов допустимы только операции отношений и операции проверки принадлежности.

Операции проверки принадлежности имеют сложные формы. В данной главе будет рассмотрена только наиболее простая форма таких операций, когда левый операнд является переменной, а правый операнд — диапазоном значений.

Обратите внимание на операции >= (больше или равно) и <= (меньше или равно). Их ни в коем случае нельзя путать с последовательностями символов => и =<. Так, символ => используется для указания на возможные альтернативы выбора в операторах case (см. гл. 4).

Далее будут приведены примеры различных выражений. Будем считать, что при этом используются следующие объявления:

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
I, J : INTEGER;
CURRENT_DAY, PREVIOUS_DAY : DAY;
```

Вот некоторые примеры правильных выражений и пояснения к ним:

Выражение	Описание выражения
I ** 5	Содержимое переменной I возводится в степень 5
3 + I * J	Значение переменной I умножается на значение переменной J и к результату прибавляется 3
5 / 3	Целое число 5 делится на 3, а дробная часть отбрасывается; результат равен 1
-I + J = 100 * J	Вначале вычисляется 100 * I, затем -I, потом -I + J; далее выполняется проверка равенства
I in 1 .. J	Выполняется проверка принадлежности. Если значение переменной I окажется в диапазоне целых чисел от 1 до J, то в результате выполнения операции получится предопределенное логическое значение TRUE, в противном случае — значение FALSE
I rem 5	Возвращает целое значение остатка от деления, знак результата совпадает со знаком делимого. Если I равно 23, то результат равен 3. Если I равно -23, то получится -3
23 rem -5	Получится 3
I mod 5	Эта операция дает число, знак которого равен знаку делителя, а абсолютное значение равно абсолютному значению остатка. Если I равно 23, то результат равен 3
I mod -5	Результат операции отрицателен. Если I равно 23, то получится -3

CURRENT\_DAY <  
PREVIOUS\_DAY

Если в списке значений для типа DAY значение переменной CURRENT\_DAY встречается раньше, чем значение, которое принимает переменная PREVIOUS\_DAY, то в результате выполнения операции получится TRUE, в противном случае — FALSE

CURRENT\_DAY not in Mon ..  
PREVIOUS\_DAY

Операция проверки принадлежности даст результат TRUE предопределенного логического типа, если значение переменной CURRENT\_DAY находится вне диапазона значений от Mon до значения, принимаемого переменной PREVIOUS\_DAY

Напомним, что в выражениях обычно не разрешается смешивать разные типы. Однако при использовании целых и перечисляемых типов можно применять атрибуты к величинам одного типа для получения значений другого типа. Например:

I + DAY'POS (CURRENT\_DAY)

— правильное выражение, поскольку значение, даваемое атрибутом POS имеет целый тип. Выражение

CURRENT\_DAY <= DAY'VAL (I)

Также правильно, если I является целым числом в диапазоне от 1 до 7, потому что значение, вырабатываемое атрибутом VAL, принадлежит к типу DAY. Разумеется, если поступать в соответствии с правилами, можно строить сложные и длинные выражения. Вот пример верного выражения:

I + J \*\* 2 - 7 / I \* J \*\* 4 + J / I \*\* 3 \* J rem I

Обратите внимание на то, что результатом выполнения операций сравнения или проверки принадлежности является значение логического типа. Выражения, в результате выполнения которых получаются значения типа BOOLEAN, можно комбинировать и далее, используя логические операции not (НЕ), and (И) и or (ИЛИ), а также некоторые другие логические операции, которые будут рассмотрены в гл. 4.

Логическая операция not является унарной и имеет такой же наивысший приоритет, как и операции \*\* и abs. Логические операции and и or — бинарные, а их приоритет является наинизшим, т.е. даже ниже, чем у операций сравнения.

Теперь повторим применение этих логических операций. Результатом вычисления выражения A and B будет TRUE тогда и только тогда, когда и результат вычисления A равен TRUE и результат вычисления B равен TRUE. Результат вычисления выражения A or B равен FALSE тогда и только тогда, когда и A и B равны FALSE. Результат вычисления выражения not A равен TRUE, если A равно FALSE, и, наоборот, результат равен FALSE, если A равно TRUE.

В языке Ада различаются динамические и статические выражения. Если возможно вычислить выражение до того, как начнется выполнение последовательности операторов программы, то выражение называется *статическим*, в противном случае — *динамическим*.

Ниже следует текст программы DAY\_CONVERSION. Входная строка для программы состоит из двух полей. В первом из них задается номер дня года в соответствии с Юлианским календарем, причем дни отсчитываются с первого января. Во втором поле указывается, на какой день недели приходится первое января нужного года. Поскольку нас могут интересовать разные года, то первое января может быть разным днем недели. Программа выводит название дня недели по представленному номеру дня в году, при этом указывается, является ли этот день выходным или нет.

### Программа DAY\_CONVERSION

```

with TEXT_IO; use TEXT_IO;
procedure DAY_CONVERSION is
  type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  package DAY_IO is new ENUMERATION_IO(DAY);
  use DAY_IO;
  JULIAN_DAY : INTEGER;
  CURRENT_DAY, FIRST_JAN_DAY : DAY;
  CURRENT_DAY_POS : INTEGER;
begin
  GET(JULIAN_DAY, 3);
  GET(FIRST_JAN_DAY);
  CURRENT_DAY_POS := JULIAN_DAY mod 7 +
                     DAY'POS(FIRST_JAN_DAY);
  if CURRENT_DAY_POS > 7
  then
    CURRENT_DAY_POS := CURRENT_DAY_POS - 7;
  and if;
  CURRENT_DAY := DAY'VAL(CURRENT_DAY_POS);
  NEW_LINE;
  PUT(" This Julian day falls on a ");
  PUT(CURRENT_DAY);
  if CURRENT_DAY not in Mon .. Fri
  then
    PUT(" is a weekend");
  else
    PUT(" is a working day ");
  end if;
end DAY_CONVERSION;

```

## 1.5. СВЕДЕНИЯ О ПАКЕТАХ, ПОДПРОГРАММАХ И ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЯХ

Программы на языке Ада составляются из одного или нескольких *программных сегментов*. Программные сегменты можно транслировать отдельно друг от друга, что создает значительные преимущества при разработке больших программных систем. Подпрограммы — это один из видов программных сегментов. Другим видом программных сегментов являются задачи (они будут рассмотрены в гл. 10) и пакеты. В этом разделе кратко рассматриваются пакеты и подпрограммы.

### 1.5.1. Пакеты

Одним из новшеств языка Ада является понятие пакета. *Пакеты* — это совокупности ресурсов, которые могут быть использованы различными программами. К этим ресурсам относятся: типы, объекты, подпрограммы, другие пакеты и различные операции, определенные для заданных типов. В дополнение к этому разработчик пакета может целиком контролировать уровень доступа и осведомленности внешнего пользователя по отношению к ресурсам пакета.

В языке Ада имеется ряд предопределенных пакетов, например пакет TEXT\_IO. Доступ к предопределенному пакету может быть обеспечен для любой программы на Аде. Операторы вида

```

with TEXT_IO;
use TEXT_IO;

```



ранее применялись нами для того, чтобы ресурсы пакета TEXT\_IO стали доступными программе на Аде. Именно это делает фраза *подключения контекста with*. Фраза *использования use* создает дополнительные удобства, позволяя употреблять в нашей программе имена, объявленные в пакете TEXT\_IO, таким образом, как будто бы они были объявлены в самой программе.

В языке Ада есть и так называемые *родовые средства* для подпрограмм и пакетов. Родовые средства Ады дают возможность программисту создавать такие подпрограммы или пакеты, которые, будучи четко ориентированы на решение конкретной задачи, имеют тем не менее достаточную степень обобщенности, чтобы охватывать и ряд вариаций этой задачи. Для использования родовых подпрограмм и пакетов в программе на Аде они должны быть *конкретизированы*. Под термином «конкретизированы» подразумевается то, что должна быть создана их уникальная версия, предназначенная для решения только заданной конкретной вариации задачи. В этой главе уже использовалась конкретизация родового пакета для создания пакета, выполняющего операции ввода-вывода с объектами конкретного целого или перечисляемого типа. Например, в нескольких из программ, приведенных в этой главе, употреблялся оператор

```
package INT_IO is new INTEGER_IO (INTEGER);
```

При помощи этого оператора из родового пакета INTEGER\_IO создавалась конкретная копия INT\_IO, а тип объектов, для которого производилась настройка родового пакета, был предопределенным целым типом INTEGER.

### 1.5.2. Подпрограммы

*Подпрограмма* состоит из двух частей – спецификации и тела. В подпрограммах из этой главы тело выполняет роль своей собственной спецификации. Имеется в виду, что ни в одной из приведенных подпрограмм нет отдельной части спецификации. Другими словами, в них нет раздельно транслируемых спецификаций. Более подробно о спецификации подпрограмм говорится в гл. 5.

В языке Ада имеются два вида подпрограмм – *процедуры* и *функции*. Оба вида подпрограмм определяют совокупность действий и являются главным способом реализации алгоритмов на Аде. На функции накладывается дополнительное требование: они должны возвращать некоторое значение, являющееся результатом каких-либо вычислений. Поэтому функции могут входить в состав выражений.

Процедуры запускаются в других программах путем указания их имени, за которым следуют список аргументов, заключенный в скобки, и точка с запятой. Обращения к процедурам считаются самостоятельными операторами.

В программах данной главы имеется много примеров вызовов процедур. Например:

```
NEW_LINE;
```

– это обращение к процедуре NEW\_LINE. Данная процедура является процедурой без параметров, и поэтому у нее отсутствуют аргументы. Другой пример вызова процедуры:

```
GET (I);
```

Здесь производится обращение к процедуре GET, переменная I служит аргументом.

Вызов функции будет выполняться тогда, когда ее имя, снабженное необходимыми аргументами, появится в выражении. Например, в пакете TEXT\_IO имеется функция LINE\_LENGTH, результатом обращения к которой является длина строки (целое число, равное количеству позиций в строке печати). Использование выражения вида LINE\_LENGTH < 80 в любой из программ данной главы приводит к вызову функции LINE\_LENGTH.

### 1.5.3. Исключительные ситуации

При выполнении программы на языке Ада всегда существует потенциальная возможность возникновения ошибки того или иного рода. Например, может оказаться неправильным тип входных данных, может встретиться попытка деления на ноль и т. д.

Эти ошибки можно обрабатывать в Аде путем возбуждения так называемых исключительных ситуаций. Если отсутствуют некоторые особые операторы, предназначенные для обработки исключительной ситуации, то *возбуждение исключительной ситуации* в главной программе на Аде обычно означает, что выполнение главной программы прекратится, и, разумеется, программа не сможет выполнить запланированные действия.

В языке Ада имеется ряд предопределенных исключительных ситуаций. Среди них — `CONSTRAINT_ERROR` (Нарушение\_уточнения), возбуждающаяся, например если значение выходит за границы диапазона, и `NUMERIC_ERROR` (Числовая\_ошибка), возникающая, например, при попытке деления на ноль.

Механизм исключительных ситуаций языка Ада может быть использован не только для обработки ошибок, возникающих при выполнении программы, но и для обработки других особых состояний. Что следует понимать под «особым состоянием», определяется программистом в зависимости от характера конкретной прикладной задачи.

Итак, выше был приведен краткий обзор таких понятий, как подпрограммы, пакеты и исключительные ситуации. Более полно эти темы освещаются соответственно в гл. 5, 7 и 11. На данном этапе пока не будем больше углубляться в эти вопросы. Теперь покажем, как, пользуясь этой отрывочной информацией, можно составлять и применять процедуры при программировании на языке Ада.

### 1.5.4. Программа, составленная с использованием процедур

В следующей программе, имеющей имя `INVENTORY_REPORT`, на вход поступают строки, имеющие тот же самый формат, что и строки для программы `INVENTORY` из разд. 1.3.4. Но здесь к входным данным предъявляется дополнительное требование: записи заранее отсортированы по признаку места расположения склада, а для каждого конкретного места расположения они отсортированы по номеру позиции товара. Сортировка по признаку места расположения склада производится не в алфавитном порядке, а в соответствии с тем местом, которое занимает это расположение в списке значений для перечисляемого типа `WAREHOUSE`. Например, `FLORIDA` идет после `NEW_YORK`. Если внутри штата имеется несколько разных складов, то входные данные могут содержать несколько записей для одного и того же номера позиции товара.

Программа должна выдавать следующие сведения. Для каждой позиции следует отобразить: номер позиции товара, описание товара, общее количество товара, имеющегося в наличии, и общее количество товара в заказе. Для каждого места расположения склада нужно напечатать общее число позиций товаров, суммарное количество имеющегося товара и суммарное количество заказанного товара (для всех позиций). В последней строке распечатки должно быть выдано итоговое количество имеющегося и заказанного товара. Если порядок следования входных строк будет нарушен, то необходимо выдать диагностическое сообщение. Строки, выпадающие из заданного порядка следования, не обрабатываются. Признаком конца данных служит строка с номером позиции товара, равным 0.

## Программа INVENTORY\_REPORT

```

with TEXT_IO; use TEXT_IO;
procedure INVENTORY_REPORT is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type QUANTITY is range 0 .. 999;
  type ITEM_NO is range 0 .. 9999;
  CURR_QUANT_ON_HAND, CURR_QUANT_ON_ORDER :
    QUANTITY;
  PREV_QUANT_ON_HAND, PREV_QUANT_ON_ORDER :
    QUANTITY;
  CURR_ITEM, PREV_ITEM : ITEM_NO;
  TOT_QUANT_ON_HAND_LOC, TOT_QUANT_ON_ORDER_LOC :
    QUANTITY;
  TOT_QUANT_ON_HAND_ITEM,
  TOT_QUANT_ON_ORDER_ITEM : QUANTITY;
  TOT_QUANT_ON_HAND, TOT_QUANT_ON_ORDER :
    QUANTITY;
  package QUANT_IO is new INTEGER_IO(QUANTITY);
  use QUANT_IO;
  package ITEM_IO is new INTEGER_IO(ITEM_NO);
  use ITEM_IO;
  CURR_ITEM_DESCRIPTION, PREV_ITEM_DESCRIPTION :
    STRING(1 .. 20);
  type WAREHOUSE is
    (ILLINOIS, NEW_YORK, TEXAS, CALIFORNIA, FLORIDA);
  package WAREHOUSE_IO is new
    ENUMERATION_IO(WAREHOUSE);
  use WAREHOUSE_IO;
  CURR_WAREHOUSE, PREV_WAREHOUSE : WAREHOUSE;

procedure NEW_ITEM_PROC is
-- Объекты, объявленные в процедуре
-- INVENTORY_REPORT, будут известны и в теле про-
-- цедуры NEW_ITEM_PROC и их можно там исполь-
-- зовать. Более детальные пояснения приведены в
-- гл.7, разд.7.3.
begin
-- Эта процедура вызывается, когда обрабатывается
-- новая позиция ведомости и когда нужно отобра-
-- зить старую позицию и итоговые данные.
  NEW_LINE;
  PUT(PREV_ITEM);
  PUT(PREV_ITEM_DESCRIPTION);
  PUT(TOT_QUANT_ON_HAND_ITEM, 5);
  PUT(TOT_QUANT_ON_ORDER_ITEM, 5);
  TOT_QUANT_ON_HAND_LOC := TOT_QUANT_ON_HAND_LOC +
    TOT_QUANT_ON_HAND_ITEM;
  TOT_QUANT_ON_ORDER_LOC := TOT_QUANT_ON_ORDER_LOC
    + TOT_QUANT_ON_ORDER_ITEM;
  TOT_QUANT_ON_HAND_ITEM := 0;
  TOT_QUANT_ON_ORDER_ITEM := 0;

```

```

PREV_ITEM := CURR_ITEM;
PREV_ITEM_DESCRIPTION := CURR_ITEM_DESCRIPTION;
PREV_QUANT_ON_HAND := CURR_QUANT_ON_HAND;
PREV_QUANT_ON_ORDER := CURR_QUANT_ON_ORDER;
end NEW_ITEM_PROC;

procedure NEW_LOC_PROC is
begin
-- Эта процедура вызывается, когда считывается
-- новое место расположения склада, а старое долж-
-- но быть отображено вместе с итоговыми данными
-- по нему.
NEW_LINE;
PUT(PREV_WAREHOUSE, 20);
PUT(TOT_QUANT_ON_HAND_LOC, 7);
PUT(TOT_QUANT_ON_ORDER_LOC, 7);
TOT_QUANT_ON_HAND := TOT_QUANT_ON_HAND +
                     TOT_QUANT_ON_HAND_LOC;
TOT_QUANT_ON_ORDER := TOT_QUANT_ON_ORDER +
                     TOT_QUANT_ON_ORDER_LOC;
TOT_QUANT_ON_HAND_LOC := 0;
TOT_QUANT_ON_ORDER_LOC := 0;
PREV_ITEM := CURR_ITEM;
PREV_ITEM_DESCRIPTION := CURR_ITEM_DESCRIPTION;
PREV_QUANT_ON_HAND := CURR_QUANT_ON_HAND;
PREV_QUANT_ON_ORDER := CURR_QUANT_ON_ORDER;
PREV_WAREHOUSE := CURR_WAREHOUSE;
end NEW_LOC_PROC;

begin
-- Вначале инициализируются некоторые итоговые
-- данные.
TOT_QUANT_ON_HAND_LOC := 0;
TOT_QUANT_ON_ORDER_LOC := 0;
TOT_QUANT_ON_HAND_ITEM := 0;
TOT_QUANT_ON_ORDER_ITEM := 0;
TOT_QUANT_ON_HAND := 0;
TOT_QUANT_ON_ORDER := 0;
GET(CURR_ITEM);
if CURR_ITEM /= 0
-- Самая первая строка используется для инициа-
-- лизации "предыдущей позиции".
then
  GET(CURR_ITEM_DESCRIPTION);
  PREV_ITEM_DESCRIPTION := CURR_ITEM_DESCRIPTION;
  GET(CURR_QUANT_ON_HAND);
  PREV_QUANT_ON_HAND := CURR_QUANT_ON_HAND;
  GET(CURR_QUANT_ON_ORDER);
  PREV_QUANT_ON_ORDER := CURR_QUANT_ON_ORDER;
  GET(CURR_WAREHOUSE);
  PREV_WAREHOUSE := CURR_WAREHOUSE;
end if;
CURR_ITEM := PREV_ITEM;
SKIP_LINE;

```

```

while CURR_ITEM /= 0
loop
  if CURR_WAREHOUSE = PREV_WAREHOUSE and
    CURR_ITEM = PREV_ITEM;
  then
    -- Заметьте, что условие истинно для первой
    -- считанной строки. Ниже производится на-
    -- копление итоговых сведений по идентичным
    -- позициям товара, хранящихся на одном и
    -- том же складе.
    TOT_QUANT_ON_HAND_ITEM :=
      TOT_QUANT_ON_HAND_ITEM +
      CURR_QUANT_ON_HAND;
    TOT_QUANT_ON_ORDER_ITEM :=
      TOT_QUANT_ON_ORDER_ITEM +
      CURR_QUANT_ON_ORDER;
  elsif CURR_WAREHOUSE = PREV_WAREHOUSE and
    CURR_ITEM > PREV_ITEM;
  then
    -- Новый вид товара на том же складе.
    NEW_ITEM_PROC;
  elsif CURR_WAREHOUSE > PREV_WAREHOUSE
  then
    NEW_ITEM_PROC;
    NEW_LOC_PROC;
  else
    PUT(" This line is bad and will be skipped ");
  end if;
  SKIP_LINE;
  GET(CURR_ITEM);
  if CURR_ITEM /= 0
  then
    -- Если номер позиции товара в строке прави-
    -- лен, то считаем с этой строки оставшиеся
    -- данные.
    GET(CURR_ITEM_DESCRIPTION);
    GET(CURR_QUANT_ON_HAND);
    GET(CURR_QUANT_ON_ORDER);
    GET(CURR_WAREHOUSE);
  end if;
end loop;
-- Достигнут конец входных данных, теперь следу-
-- ет отобразить последние итоговые сведения по
-- последнему виду товара на складе. Это делает-
-- ся в последних двух строках.
NEW_ITEM_PROC;
NEW_LOC_PROC;
NEW_LINE;
PUT(" THE GRAND TOTAL IS ");
PUT(TOT_QUANT_ON_HAND);
PUT(TOT_QUANT_ON_ORDER);
end INVENTORY_REPORT;

```

В программе INVENTORY\_REPORT используется зарезервированное слово `elsif` (иначе\_если). Оно входит в состав оператора `if` (если). Вначале вычисляется условие, стоящее за зарезервированным словом `if`. Если условие истинно, то выполняются операторы, следующие за зарезервированным словом `then` (то). В противном случае вычисляется выражение, стоящее за зарезервированным словом `elsif`. Если оно в свою очередь истинно, то выполняются операторы, стоящие за `elsif ... then`. В противном случае вычисляется следующее условное выражение. Этот процесс продолжается до тех пор, пока одно из условий, расположенных за зарезервированным словом `elsif`, не окажется истинным, либо пока не будет достигнуто зарезервированное слово `else` (иначе). В последнем случае будут выполняться операторы, следующие за зарезервированным словом `else`.

Приведенный ниже пример поможет лучше уяснить употребление ветви `elsif` в операторе `if`. Здесь предполагается, что переменная `I` принадлежит к типу `INTEGER`.

```

if I > 3
then
  I := I * I;
elsif I > 1
then
  I := I * 3;
elsif I > -3
then
  I := 0;
else
  I := I / 3;
end if;

```

Если в данном примере `I` равно 4, то истинно первое условие (`I > 3`). При этом значение `I` станет равным  $4 \times 4 = 16$ . Другие условия в этом случае не проверяются (хотя они также истинны поскольку  $4 > 1$  и  $4 > -3$ ), так как ранее проверенное условие уже оказалось истинным. Условие `I > 1` проверяется тогда, когда условие `I > 3` не выполняется. А условие `I > -3` проверяется только тогда, когда и `I > 3`, и `I > 1` ложны. Операторы, следующие за зарезервированным словом `else`, выполняются тогда и только тогда, когда все проверенные ранее условия (`I > 3`, `I > 1`, `I > -3`) ложны.

## УПРАЖНЕНИЯ

1. Измените программу MAXALL из разд. 1.1.3 так, чтобы в ней определялись наибольшее число и число, следующее за ним.
2. Перепишите программу CHAR\_MAX3 из разд. 1.3.2 таким образом, чтобы печаталась первая по алфавиту буква среди нескольких введенных букв. Признаком конца последовательности букв служит символ `"!"`.
3. Переделайте программу HEAVY из разд. 1.3.2 так, чтобы она обрабатывала дополнительное поле строки в колонке 21. Там размещается число торговых дней по понедельникам, за которыми не следуют торговые дни по вторникам, и количество торговых дней по вторникам, перед которыми не идут торговые дни по понедельникам.
4. Модифицируйте программу UP\_MONDAY из разд. 1.3.4 таким образом, чтобы в ней в добавление к уже представляемым сведениям печаталось число торговых дней по понедельникам, за которыми не следуют торговые дни по вторникам, и количество торговых дней по вторникам, перед которыми не идут торговые дни по понедельникам.
5. Внесите такие изменения в программу INVENTORY из разд. 1.3.4, чтобы выводился

ответ на вопрос: верно ли, что на складе в Техасе зарегистрировано больше наименований товаров, чем в Калифорнии?

6. Измените программу DAY\_CONVERSION из разд. 1.4 так, чтобы можно было получить ответ на вопрос: правда ли, что на вход программы подано больше строк с номерами дней, попадающих на понедельники, чем на вторники?

7. Перепишите программу INVENTORY\_REPORT из разд. 1.5.4 в предположении, что имеется только одно место расположения склада. Данные о видах товаров, хранящихся на складе, отсортированы по номерам позиций. Допускается наличие строк с одинаковыми номерами позиций товаров. Эту программу написать легче, чем исходную, поскольку количество уровней размещения товаров тут уменьшается на единицу.



## Глава 2

# Действительные, регулярные и комбинированные типы

## 2.1. ДЕЙСТВИТЕЛЬНЫЕ ТИПЫ

*Действительные типы* обеспечивают конечный набор значений, аппроксимирующих действительные числа. Согласно математическим представлениям, существует бесконечно много действительных чисел и они могут быть произвольно близки друг к другу. Но в ЭВМ для представления действительного числа предусмотрено только конечное количество бит. По этой причине она не может обеспечить абсолютно точное представление любого действительного числа и поэтому в языке Ада для каждого из объявляемых типов, в которых используются действительные числа, определяется так называемый *модельный набор действительных чисел*.

Как и в случае целых чисел, здесь имеется предопределенный действительный тип, имеющий название FLOAT (Плавающий).

Например:

```
RATE: FLOAT;
```

объявляет переменную RATE как переменную действительного типа FLOAT. Однако при использовании этого объявления пользователь полагается на такую точность, которая определяется используемой в конкретном случае аппаратурой.

Для того чтобы передать управление точностью от аппаратных средств к программным (такая передача улучшает мобильность программ), в языке Ада имеется механизм для определения границ ошибок представления для значений конкретного действительного типа. Этот механизм описывается в следующих разделах книги. В языке Ада могут быть определены два вида действительных типов – типы с плавающей точкой (или плавающие типы) и типы с фиксированной точкой (или фиксированные типы). *Плавающие типы* обеспечивают представление действительных значений с некоторой относительной точностью, а фиксированные типы представляют действительные значения с некоторой абсолютной точностью. Итак: механизм точности в языке Ада различен для действительных типов. Целые и действительные типы (плавающие и фиксированные) в совокупности называются *числовыми типами*.

### 2.1.1. Типы с плавающей точкой (плавающие типы)

Объявление типа с плавающей точкой производится в соответствии со схемой

```
type имя_типа is digits статическое_целое_выражение;
```

Если задается диапазон значений, то схема приобретает вид

```
type имя_типа is digits статическое_целое_выражение range L..R;
```

Член объявления “digits статическое\_целое\_выражение” называется *указанием погрешности представления* (accuracy constraint). Вспомните, что статическое целое выражение – это выражение, значение которого может быть вычислено до начала выполнения программы. Член “range L..R” называется *уточнением диапазона значений*

(range constraint). Величина L определяет нижнюю (левую) границу этого диапазона, а величина R — верхнюю (правую) границу.

Примеры объявлений плавающих типов:

type DISTANCE is digits 8;

type SECURITY\_PRICE is digits 10 range 0.0..10000.00;

В первом объявлении вводится тип DISTANCE с относительной точностью в 8 десятичных цифр. Каждая десятичная цифра соответствует приблизительно 3.32 двоичным разрядам, поэтому точность, выраженная в двоичной системе счисления, здесь составит  $8 \times 3.32$  или 27 двоичных разрядов. Обозначим это число через B.

Определено, что для заданной точности B (в двоичной системе) порядок модельных чисел может находиться в пределах от  $-4 * B$  до  $+4 * B$ ; набор модельных чисел всегда включает нулевое значение. Общая форма модельных чисел такова:

знак \* двоичная\_мантисса \*  $2.0^{**}$  порядок

где двоичная\_мантисса имеет B двоичных цифр после десятичной точки (27 в рассмотренном случае). Одно и то же число может иметь множество допустимых внешних представлений, соответствующих приведенной выше общей форме. Например, число 0.5 (0.1 в двоичной системе) можно записать как

$1/2 * 2^{**}0$

$1/4 * 2^{**}1$

и т. д.

Модельные числа *нормализованы*. Это значит, что среди множества возможных представлений двоичного числа, соответствующих общей форме, выбирается число, двоичная мантисса которого равна по меньшей мере 0.1 в двоичной или 0.5 в десятичной системе счисления.

Наименьшее ближайшее к нулю положительное число среди модельных чисел выражается формулой:

$2.0^{**}(-4 * B - 1)$

Оно равно значению 0.1 в двоичной или 0.5 в десятичной системе, умноженному на 2 в максимальной отрицательной степени:

$0.5 * 2^{**}(-4 * B)$

Для нашего примера при  $B = 27$  это число равно  $2.0^{**}(-109)$ . Оно определяется отдельно для каждого плавающего типа. Его можно получить при помощи запроса атрибута DISTANCE'SMALL. Если F — произвольный плавающий тип, то форма записи — F'SMALL.

Наибольшее положительное модельное число для нашего типа DISTANCE будет иметь двоичную мантиссу 0.11... (27 единиц)... 11, которую можно также записать как 1-0.00... (26 нулей)... 01. Показатель степени равен  $4 * 27 = 108$ . Само число равно

$(1 - 2^{**}(-27)) * 2.0^{**}(4 * 27)$

Это значение можно получить при помощи запроса атрибута DISTANCE'LARGE. В общем случае наибольшее положительное значение для произвольного плавающего типа F можно получить при запросе атрибута F'LARGE. Это наибольшее число равно

$(1 - 2^{**}(-B)) * 2.0^{**}(4 * B)$

Другой важной величиной при работе с модельными числами служит разность между 1.0 и следующим модельным числом. Для типа DISTANCE — это разность между  $0.1 * 2^{**}(1)$  и  $0.100... (25 \text{ нулей})... 001 * 2^{**}(1)$ , которая равна  $2^{**}(1 - 27)$ . Это

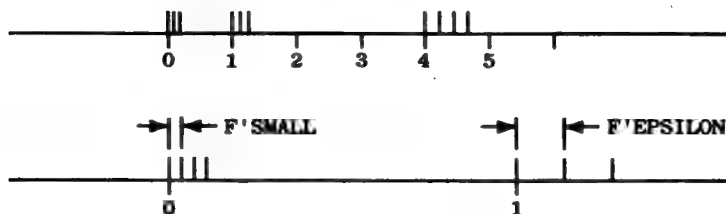


Рис. 2.1. Модельные числа для плавающих типов.

значение может быть получено при запросе атрибута `DISTANCE'EPSILON`. В общем случае для любого плавающего типа `F` разность между 1.0 и следующим большим модельным числом равна  $2^{**}(1 - B)$ . Ее можно получить посредством запроса `F'EPSILON`.

Как говорилось выше, для плавающих типов задается относительная погрешность представления. Количество значащих цифр у плавающих чисел остается постоянным (например, для типа `DISTANCE`—это 8 десятичных цифр, что соответствует 27 двоичным цифрам), а разность между двумя соседними числами изменяется. Она минимальна в окрестности нуля и равна  $2^{**}(-1 - 4 * B)$ . В окрестности единицы она становится в  $2^{**}(3 * B + 2)$  раз больше. В этом случае она равна  $2^{**}(1 - B)$ . А разность между наибольшим модельным числом и числом, следующим за ним, равна  $2^{**}(3 * B)$ . Это иллюстрирует рис. 2.1.

Во время вычислений может получиться значение, не являющееся модельным числом. Если значение окажется в интервале между двумя соседними модельными числами, то будет выбрано одно из них. Какое именно модельное число будет взято, зависит от конкретной реализации Ады.

Когда в объявлении плавающего типа присутствуют и указание погрешности представления, и уточнение диапазона значений, то часть модельных чисел, попадающих внутрь заданного диапазона, сохраняется, а остальные числа отбрасываются. Обратите внимание на то, что левая и правая границы диапазона значений должны задаваться статическими действительными выражениями.

В нижеследующем списке операции для плавающих типов перечислены в порядке убывания их приоритетов:

<code>abs, **</code>	Вычисление абсолютного значения, возведение в степень
<code>*, /</code>	Умножение, деление
<code>+, -</code>	Знаки (унарные операции)
<code>+, -</code>	Сложение, вычитание (бинарные операции)
<code>=, /=, &lt;, &lt;=, &gt;, &gt;=, in, not in</code>	

Заметим, что операция `**` (возведение в степень) определена только для возведения плавающего числа в целую степень. Результат выполнения арифметической операции с объектами плавающего типа будет принадлежать к тому же типу.

### 2.1.2. Фиксированный тип

Можно объявить действительный тип, имеющий абсолютную погрешность представления, если воспользоваться представлением с фиксированной точкой. Форма объявления для фиксированного типа такова:

`type имя_типа is delta абсолютная_погрешность_представления range L .. R;`

Вот некоторые примеры:

```
type AMOUNT is delta 0.001 range 0.00..10000.00;
type PRICE is delta 0.0025 range 0.00..200.00;
type LEVEL is delta 0.25 range -100.00..5000.00;
```

Обратите внимание на то, что при объявлении фиксированных типов всегда нужно задавать уточнение диапазона значений, в то время как для плавающих типов указание диапазона не обязательно.

Модельные числа для действительных чисел с фиксированной точкой отстоят друг от друга на одинаковом расстоянии, что обеспечивает заданную абсолютную точность представления. Гарантированная разность между двумя соседними числами не превышает требуемой абсолютной погрешности, которая задается в выражении, стоящем после зарезервированного слова *delta*. Это выражение должно быть статическим, а результат его вычисления — положительным действительным числом. Вполне возможно, что при реализации модельных чисел транслятор выберет другую величину погрешности, вероятно, более удобную для представления на конкретной ЭВМ. В этом случае фактическая абсолютная погрешность не может превышать указываемую программистом величину погрешности, а разность между любыми двумя соседними модельными числами должна быть равна фактической погрешности, умноженной на некоторый постоянный коэффициент.

Несколько неожиданной особенностью фиксированных типов может явиться то, что левая (L) и правая (R) границы диапазона значений, определяемые действительными статическими выражениями, *не обязаны* входить во множество модельных чисел. Единственное требование к ним состоит в том, что последовательность модельных чисел начинается (для L) и заканчивается (для R) на расстоянии от L и R (соответственно), не превышающем абсолютной погрешности заданной в объявлении типа. Так, например, если в объявлении типа LEVEL (см. выше) указано  $L = -100.00$ , то нет гарантии, что величина  $-100.00$  действительно попадет во множество модельных чисел. Единственное, что можно гарантировать: наименьшее модельное число отстоит от  $-100.00$  не более чем на  $\pm 0.25$ .

Запрос атрибута F'FIRST, где F — фиксированный тип, дает возможность получить и в дальнейшем использовать значение наименьшего модельного числа. Для нашего примера величина LEVEL'FIRST должна отстоять от  $-100.00$  не более чем на  $\pm 0.25$ . Сходным образом, значение наибольшего модельного числа можно получить при запросе атрибута F'LAST. Для типа LEVEL атрибут LEVEL'LAST даст значение модельного числа, находящегося в пределах  $\pm 0.25$  от 5000.00.

Остальные атрибуты фиксированных типов достаточно ясны. Если F — фиксированный тип, то запрос атрибута F'DELTA даст значение указываемой программистом абсолютной погрешности. Для приведенного выше примера объявления фиксированных типов можно воспользоваться атрибутом PRICE'DELTA, который даст значение 0.0025, а атрибут AMOUNT'DELTA даст значение 0.001.

Приведем и некоторые другие атрибуты для фиксированных типов:

Атрибут	Описание
F'SMALL	Наименьшее положительное модельное число для типа F
F'LARGE	Наибольшее положительное модельное число для типа F

При работе с фиксированными типами возникают ошибки округления. Поскольку фиксированные типы наиболее часто используются при решении экономических задач, то встает резонный вопрос: каким образом можно обеспечить то, чтобы ошибки округления не приводили к получению неприемлемых, скажем, для бухгалтера результатов? Что делать, если при сложении одного пенса с одним пенсом получится три

пенса? Одним из способов уменьшения вероятности таких ситуаций может служить выбор достаточно малой абсолютной погрешности представления фиксированных чисел. Скажем, при работе с пенсами следует выбрать величину абсолютной погрешности, равную 0.0001.

В программах необходимо выполнять ввод-вывод чисел с фиксированной точкой. Для записи и для чтения значений некоторого фиксированного типа, как и в случаях целых и перечисляемых типов, требуются специальные пакеты, которые являются частью пакета TEXT\_IO. То же самое справедливо и для плавающих типов. Использование таких пакетов будет продемонстрировано позже в программе PAYROLL.

Операции с фиксированными типами включают умножение (\*) и деление (/). В этих операциях участвуют два фиксированных числа одного и того же типа, а в результате их выполнения получается число так называемого *универсального фиксированного типа*, имеющего точность, неявно определяемую реализацией языка Ада. Число универсального фиксированного типа следует явно преобразовать в число требуемого типа. Для этого нужно выражение универсального фиксированного типа заключить в скобки, а перед ними поставить имя желаемого типа. Пусть, например, имеется объявление

```
AMT_IN, AMT_OUT, AMT_RATIO: AMOUNT;
```

Нам необходимо получить значение отношения  $AMT\_IN/AMT\_OUT$ , которое относится к универсальному фиксированному типу. На Аде это записывается так:

```
AMT_RATIO := AMOUNT (AMT_IN/AMT_OUT);
```

При сложении и вычитании не требуется явного преобразования типов. Поэтому можно написать

```
AMT_IN := AMT_IN + AMT_OUT;
```

Для объектов фиксированного типа (переменных и констант) операция возведения в степень не определена.

Ниже приводится сводка операций для фиксированных типов, начиная с наивысшего приоритета:

abs	Абсолютное значение
*, /	Умножение, деление
+, -	Знаки (унарные операции)
+, -	Сложение, вычитание (бинарные операции)
=, /=, <, <=, >, >=, in, not in	

Как для плавающих, так и для фиксированных типов результат вычисления выражения должен находиться в пределах специфицированного для типа диапазона значений<sup>1)</sup>.

### 2.1.3. Программа, в которой используются действительные типы

В нижеследующей программе PAYROLL (платежная ведомость) иллюстрируется применение действительных типов. Программа выполняет считывание информации о служащих. Каждая строка содержит сведения только об одном служащем и имеет следующий формат:

<sup>1)</sup> Однако промежуточные значения могут выходить за этот диапазон.—Прим. перев.

Позиция	Данные
1-9	Номер по социальному страхованию
10-30	Фамилия
31-35	Количество часов, отработанных за неделю (два знака до и после десятичной точки, например 42.35)
36-40	Почасовая оплата (в долларах и центах, например 12.45)
41-42	Количество иждивенцев (целое число, например 5)

Признаком конца файла служит строка с номером по социальному страхованию, равным 999999999.

Для каждой считанной строки с данными о служащем будут выведены: фамилия служащего; номер по социальному страхованию; номинальная заработная плата за неделю; сумма, выдаваемая на руки. Будем считать, что налог на социальное страхование (обозначается как FICA) составляет 7.0% от номинальной зарплаты. Сумма, с которой исчисляется подоходный налог, равна номинальной плате минус 10 долл., умноженные на число иждивенцев. Федеральный налог составляет 25% от этой суммы, а налог штата - 5% от нее. Сумма, выдаваемая служащему на руки, определяется как разница между номинальной зарплатой и налоговыми выплатами.

### Программа PAYROLL

```
with TEXT_IO; use TEXT_IO;
procedure PAYROLL is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type PAY is delta 0.001 range 0.00 .. 10000.00;
  HOURS_WORKED, HOURLY_RATE : PAY ;
  GROSS_PAY, NET_PAY, TAX_INCOME : PAY;
  NO_DEPENDENTS : INTEGER ;
  SOC_SEC_NO : STRING (1 .. 9 );
  EMP_NAME : STRING (1 .. 21);
  FICA_RATE : constant PAY := 0.07;
  FED_RATE : constant PAY := 0.25;
  STATE_RATE: constant PAY := 0.05;
  package PAY_IO is new FIXED_IO (PAY);
  -- Здесь делаются доступными процедуры
  -- GET и PUT, выполняющие ввод-вывод
  -- объектов типа PAY.
  -- См. ниже их использование и формат.
  use PAY_IO;
begin
  GET(SOC_SEC_NO);
  while SOC_SEC_NO /= "999999999"
  loop
    GET(EMP_NAME);
    GET(HOURLY_RATE,5); -- Во вводимом
    -- числе должна быть десятичная точка.
    GET(HOURS_WORKED,5); -- Во вводимом
    -- числе должна быть десятичная точка.
    GET(NO_DEPENDENTS,2);
    GROSS_PAY := PAY(HOURLY_RATE * HOURS_WORKED);
    -- Запомните: умножение дает результат универ-
```

```

-- сального фиксированного типа, его надо
-- преобразовать к типу PAY.
if PAY ( NO_DEPENDENTS * 10 ) < GROSS_PAY
then
    TAX_INCOME := GROSS_PAY -
                    PAY ( NO_DEPENDENTS * 10 ) ;
else
    TAX_INCOME := 0.00;
end if;
NET_PAY := GROSS_PAY - PAY(GROSS_PAY * FICA_RATE)
            - PAY ( TAX_INCOME * FED_RATE )
            - PAY ( TAX_INCOME * STATE_RATE ) ;

NEW_LINE;
PUT(SOC_SEC_NO);
PUT(EMP_NAME);
PUT(GROSS_PAY,7,2);
-- Этот оператор - одна из версий процедуры GET
-- из пакета PAY_10, где 7 позиций отводится
-- под целую часть и 2 - под дробную.
PUT(NET_PAY,7,2);
SKIP_LINE;
GET(SOC_SEC_NO);
end loop;
end PAYROLL ;

```

Обратите внимание на то, что для числовых типов разрешены явные преобразования типов. Преобразование от одного типа к другому будет выполнено, если перед выражением, заключенным в скобки, поставить имя требуемого типа. Вот несколько примеров преобразования типов с использованием объявлений из программы PAYROLL:

Преобразование	Результат
INTEGER (GROSS_PAY)	Значение GROSS_PAY округляется до ближайшего целого
FLOAT (21)	Целое число 21 преобразуется к предопределенному типу FLOAT
PAY (NO_DEPENDENTS)	Целое значение NO_DEPENDENTS преобразуется к модельному числу фиксированного типа PAY

Целые, перечисляемые и действительные типы в совокупности называются *скалярными типами*, поскольку значения этих типов не имеют компонент. В следующих разделах будут представлены типы, объекты которых состоят из нескольких логических частей. Это — регулярные типы и комбинированные типы.

## 2.2. РЕГУЛЯРНЫЕ ТИПЫ

*Регулярный тип* представляет собой совокупность логически связанных компонент. Объекты регулярного типа называются *массивами*. Каждая *компонента* принадлежит к одному и тому же типу и занимает в массиве единственное в своем роде положение по отношению к прочим компонентам. Позиция компоненты задается одним или несколькими индексами. Если есть один индекс, то массив называется *одномерным*. Если же у массива имеются два или более индексов, то он называется *многомерным*. В языке Ада отсутствует предопределенное ограничение на число индексов у массива. Понятие массива иллюстрирует рис. 2.2. На рис. 2.2,*a* показан одномерный массив типа STATE\_RATE. На рис. 2.2,*b* изображен двумерный массив типа SHIPPING\_RATES. Эти массивы взяты из программы, приведенной в данном разделе.



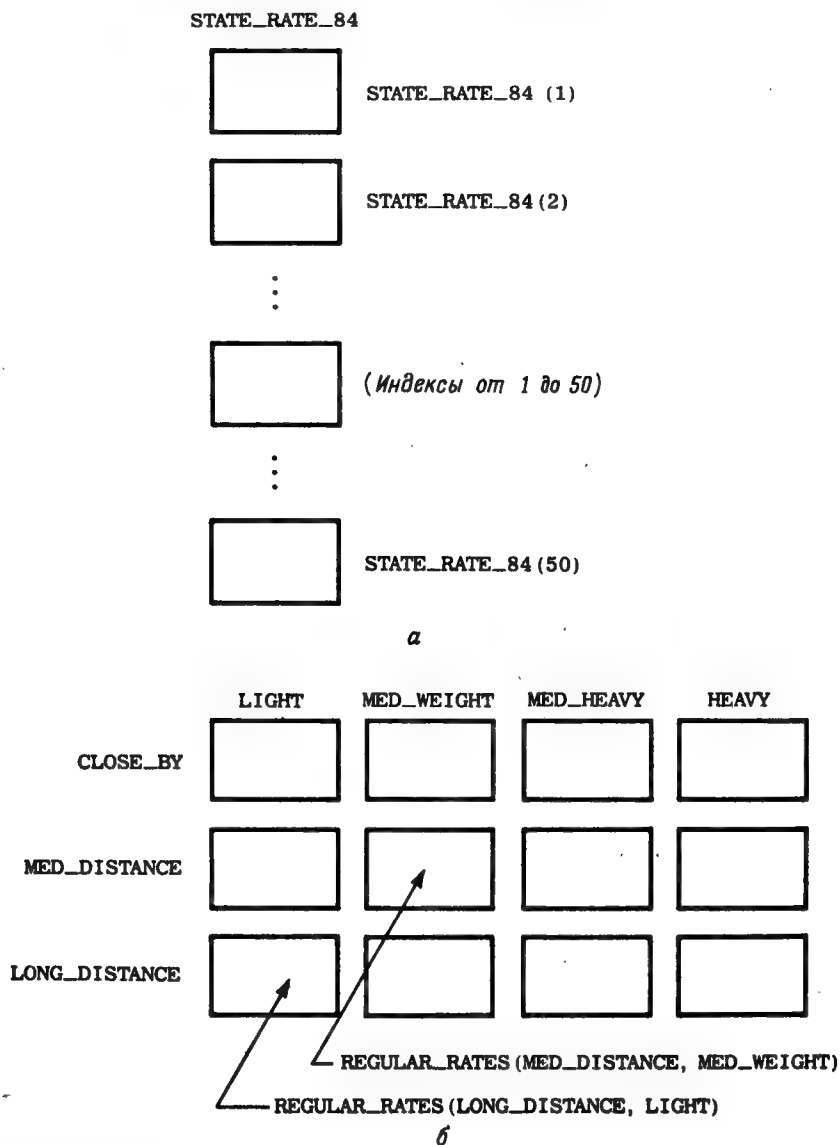


Рис. 2.2. Массивы.

*a* – переменная STATE\_RATE\_84 типа STATE\_RATE; *b* – переменная REGULAR\_RATES типа SHIPPING\_RATES.

В языке Ада есть две разновидности регулярных типов: регулярные типы с уточненными и с неуточненными границами диапазонов индексов. Для *неуточненных регуляторных типов* эти границы, т. е. начальное и конечное значения каждого индекса, не указываются в объявлении типа. Вместо этого границы задаются при объявлениях различных объектов (переменных и констант) этого типа. Фактические же значения границ могут быть заданы еще позже – во время выполнения программы.

Для *уточненных регулярных типов* границы диапазонов индексов должны быть известны к моменту объявления объектов (переменных и констант), принадлежащих к

этим типам. Как следствие этого, уточненные регулярные переменные (т.е. массивы) одного и того же типа имеют одинаковые границы, в то время как неуточненные регулярные переменные одинакового типа могут иметь разные границы.

*Границы диапазонов индексов* у массивов специфицируются выражениями, результат вычисления которых имеет дискретный (т.е. целый или перечисляемый) тип. Если все выражения статические, т.е. их значения могут быть вычислены во время трансляции программы, то массив называется *статическим*. Массив называется *динамическим*, если по крайней мере одна из границ задается динамическим выражением. Динамическим называется такое выражение, значение которого может быть определено только во время выполнения программы. Как уточненные, так и неуточненные регулярные переменные могут представлять собой динамические массивы.

Ниже приводится ряд примеров, разъясняющих введенные понятия.

### 2.2.1. Уточненные регулярные типы

Объявление уточненных регулярных типов имеет вид

```
type имя_типа is array уточнение_диапазонов_индексов of тип_компонент;
```

Уточнение диапазонов индексов — это заключенная в скобки последовательность диапазонов индексов, разделенных запятыми. Диапазоны должны быть дискретными, т.е. их левые и правые границы должны задаваться выражениями целого или перечисляемого типа.

**Примеры.** Ниже приводятся объявления регулярных типов с уточненными границами и объекты этих типов:

```
type STATE_RATES is array (1..50) of FLOAT;
```

Здесь имя\_типа — это STATE\_RATES, а уточнение диапазонов индексов — это (1..50), определяющее 50 компонент. Каждая компонента имеет тип FLOAT. Диапазон изменения единственного индекса лежит в пределах от 1 до 50.

```
STATE_RATE_84, STATE_RATE_85: STATE_RATES;
```

Здесь переменные STATE\_RATE\_84 и STATE\_RATE\_85 имеют тип STATE\_RATES, и каждая переменная состоит из 50 компонент.

```
N: INTEGER;
type CLASS_STUDENT_ID is array (1..N) of INTEGERS;
SECTION_A, SECTION_B: CLASS_STUDENT_ID;
```

Здесь SECTION\_A и SECTION\_B — примеры динамических массивов. Остальные массивы — статические.

```
type WAREHOUSE is (CHICAGO, NEW_YORK, FRESNO, BOSTON);
```

Здесь объявлен перечисляемый тип, который будет диапазоном изменения индексов для регулярного типа LOCATIONS

```
type LOCATIONS is array (WAREHOUSE) of FLOAT;
CURR_LOCATION, PREV_LOCATION: LOCATIONS;
type DISTANCE_CLASS is (CLOSE_BY, MED_DISTANCE,
LONG_DISTANCE);
type WEIGHT_CLASS is
(LIGHT, MED_WEIGHT, MED_HEAVY, HEAVY);
type PRICES is delta 0.0001 range 0.000 ..
5000.000;
type SHIPPING_RATES is array (DISTANCE_CLASS,
WEIGHT_CLASS) of PRICES;
```

Здесь регулярный тип `SHIPPING_RATES` имеет два диапазона изменения индексов, которые являются частями уточнения диапазонов индексов. Это – двумерный массив.

В качестве заключительного примера приведем объявление:

`REGULAR_RATES, PREMIUM_RATES: SHIPPING_RATES;`

Можно обратиться к компоненте любого объекта регулярного типа, если вслед за именем объекта поместить заключенное в скобки необходимое значение индекса. Вот некоторые примеры.

`STATE_RATE_84 (48)`

Компонента 48 массива `STATE_RATE_84` имеет тип `FLOAT`

`CURR_LOCATION (CHICAGO)`

Компонента `CHICAGO` массива `CURR_LOCATION` имеет тип `FLOAT`

`REGULAR_RATES (CLOSE_BY, HEAVY)`

Компонента типа `PRICES` имеет два индекса: индекс `GLOSE_BY` относится к типу `DISTANCE_CLASS`, индекс `HEAVY` относится к типу `WEIGHT_CLASS`

Значения индексов могут быть получены с помощью вычисления любого выражения. Результат вычисления должен иметь требуемый тип и находиться в заданных пределах.

Дискретный диапазон, т. е. множество значений целого или перечисляемого типа, может вообще не содержать никаких значений. В этом случае диапазон называется пустым, а массив вовсе не имеет компонент. Например, предположим, что величина `N` в одном из предыдущих примеров инициализирована так:

`N: INTEGER := -1;`

Тогда массивы `SECTION_A` и `SECTION_B` будут иметь пустой диапазон.

Использование массивов будет показано на примере программы `SHIP_RATE`. Но, перед тем как будет приведена программа, введем новый оператор языка Ада, который облегчит эффективную работу с массивами. Это – второй вид оператора цикла (`loop statement`), называемый циклом *для* (`for loop`). Ранее уже был рассмотрен цикл *пока* (`while loop`).

## 2.2.2. Циклы `for`

Общая форма оператора цикла *для* (`for loop statement`) такова:

```
for параметр_цикла in дискретный_диапазон
  loop
    последовательность_операторов
end loop;
```

Вместо зарезервированного слова `in` (в) в операторе цикла `for` можно употребить слова `in reverse` (в обратном порядке).

Параметр цикла – это переменная, тип которой – такой же, как и тип у дискретного диапазона. Поэтому множество допустимых значений параметра цикла ограничено величинами, указанными в этом диапазоне. Параметр цикла не следует объявлять. Его значение за пределами оператора цикла не определено. Объявление параметра цикла произойдет автоматически при указании его в операторе цикла. Такой вид объявления называется *неявным объявлением*. Неявные объявления – это исключение из общего правила, которое гласит, что все переменные необходимо объявлять в декларативной части подпрограммы.

Член последовательность операторов, входящий в состав оператора цикла, выполняется по одному разу для каждого из значений, входящих в дискретный диапазон цикла. При этом значения параметра цикла из дискретного диапазона устанавливаются следующим образом: если присутствует только одно зарезервированное слово `in`, то параметр цикла принимает значения из дискретного диапазона в порядке возрастания, начиная с нижней границы диапазона; если же в оператор цикла входят зарезервированные слова `in reverse`, то параметр цикла будет принимать значения из дискретного диапазона в порядке убывания, начиная с верхней границы диапазона. Если диапазон пуст, то последовательность операторов не выполнится ни разу.

Как и можно было бы предположить, параметр цикла нельзя располагать слева от символа `:=` в операторе присваивания. Его нельзя также изменять каким-либо иным способом внутри цикла. Другими словами, параметр цикла предназначен для обеспечения вычисления цикла при заданных значениях и в заданном порядке. Изменение параметра цикла привело бы к нарушению этих условий.

**Пример.** Рассмотрим некоторые примеры циклов `for`. Вот один из них:

```
J := 0;
for I in 1 .. 20
  loop
    J = J + 1;
  end loop;
```

Здесь считается, что `J` относится к целому типу. Этот цикл выполняет суммирование целых чисел от 1 до 20, а результат присваивается переменной `J`. Поскольку тип выражений, задающих границы дискретного диапазона — целый, то и неявно объявляемый параметр цикла `I` также принадлежит к целому типу.

Другой пример:

```
for I in 0 .. -1
  loop
    J := J * 2;
  end loop;
```

Здесь диапазон — пустой<sup>1)</sup>, и оператор

```
J := J * 2;
```

не выполнится ни разу.

Заключительный пример:

```
GET (K,3);
J := 0;
for I in -K ** 3 .. K ** 2
  loop
    J = I ** 2 + J/2;
  end loop;
```

Предполагается, что `K` — переменная целого типа. Заметьте, что при отрицательном `K` диапазон будет пустым.

Как упоминалось ранее, можно писать программы, употребляя только циклы `while`. В случае программы `SHIP_RATE`, текст которой дан ниже, применение циклов `for` значительно улучшает ее читабельность. Вообще говоря, циклы `for` хорошо подходят для последовательной обработки компонент массива, так как в этом случае

<sup>1)</sup> Диапазон будет пустым, если верхняя его граница имеет меньшее значение, чем нижняя. — *Прим. перев.*

для каждой из компонент выполняются аналогичные операторы. Параметр цикла обеспечивает удобный и хорошо контролируемый способ индексации нужных компонент и действий с ними.

### 2.2.3. Программа, в которой употребляются массивы

В нижеследующей программе вначале строится таблица тарифов на доставку грузов. Она заполняется значениями из первых трех строк входного файла. Остальные строки файла содержат информацию об отправляемых грузах. В таблице – три строки и четыре столбца. Строки таблицы соответствуют трем классам расстояния, а столбцы – четырем классам веса груза. Каждая из первых трех строк входного файла содержит по четыре действительных числа, представляющих собой тарифы на оплату доставки фунта груза. Таким образом, всего может быть до 12 различных категорий грузов. В остальных строках файла размещается информация о номере позиции груза, о его весе и о расстоянии, на которое его следует доставить. Признаком конца данных служит строка, в которой номер позиции груза равен 9999999. Каждое входное число занимает семь позиций строки. Во входном файле насчитывается не менее трех строк. Программа должна вывести стоимость доставки для каждой представленной позиции отправляемого груза.

#### Программа SHIP\_RATE

```
with TEXT_IO; use TEXT_IO;
procedure SHIP_RATE is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type ALLOWED_WEIGHT is digits 10 range 0.00 ..
                                7000.00;
  -- Следует проверить, какое макс. количество
  -- цифр здесь допустимо для данной версии языка.
  type ALLOWED_DISTANCE is digits 10 range 0.00 ..
                                8000.00;
  -- Здесь объявлены два плавающих типа, для к-рых
  -- заданы диапазон значений и кол-во цифр (10).
  package DISTANCE_IO is new
                                FLOAT_IO(ALLOWED_DISTANCE);
  use DISTANCE_IO;
  package WEIGHT_IO is new FLOAT_IO(ALLOWED_WEIGHT);
  use WEIGHT_IO;
  -- Здесь обеспечивается доступ к пакетам, нужным
  -- для чтения и записи некоторых плавающих типов.
  type DISTANCE_CLASS is (CLOSE_BY, MED_DISTANCE,
                           LONG_DISTANCE);
  -- Это – объявление перечисляемого типа. Он
  -- используется ниже для индексации массивов.
  CURR_DIST_CLASS : DISTANCE_CLASS ;
  -- Это – объявление переменной, принадлежащей
  -- к перечисляемому типу DISTANCE_CLASS.
  type WEIGHT_CLASS is
    ( LIGHT, MED_WEIGHT, MED_HEAVY, HEAVY);
  CURR_WEIGHT_CLASS : WEIGHT_CLASS;
  type DIST_CATEGORIES is array ( DISTANCE_CLASS )
    of ALLOWED_DISTANCE;
  -- Это – объявление одномерного массива, индекс-
  -- сация которого выполняется с помощью
```

```

-- перечисляемого типа DISTANCE_CLASS.
ACT_DIST_LIM : constant DIST_CATEGORIES :=
  ( 100.00, 400.00, 8000.00 ) ;
-- Это об'явление означает, что
-- ACT_DIST_LIM(CLOSE_BY) равно 100.00, ... ,
-- ACT_DIST_LIM(LONG_DISTANCE) равно 8000.00.
-- Присваивание числовых значений массиву
-- можно выполнить, если задать в скобках
-- список значений компонент, называемый агре-
-- гатом. Значения агрегатов можно присваивать
-- как переменным, так и константам, например
-- ACT_DIM_LIST. Другие возможные способы
-- определения агрегатов обсуждаются в гл.4.
type WEIGHT_ARRAY is array ( WEIGHT_CLASS ) of
  ALLOWED_WEIGHT;
ACT_WEIGHT_LIM : constant WEIGHT_ARRAY :=
  ( 10.00, 20.00, 40.00, 5000.00 ) ;
-- Об'явлен массив-константа типа WEIGHT_ARRAY,
-- определяющий верхние границы категорий веса.
-- Здесь опять употреблен агрегат.
type PRICES is delta 0.0001 range 0.000 ..
  5000.00;
-- Об'явлен фиксированный тип.
package PRICES_IO is new FIXED_IO(PRICES);
use PRICES_IO;
-- Сделан доступным пакет, необходимый для ввода-
-- вывода величин фиксированного типа PRICES.
type SHIPPING_RATES is array ( DISTANCE_CLASS,
  WEIGHT_CLASS ) of PRICES;
-- Этот тип определяет двумерный массив с тремя
-- строками и четырьмя столбцами, индексируемый
-- с помощью об'ектов перечисляемых типов.
CURR_RATES : SHIPPING_RATES ;
ITEM_NO : INTEGER ;
ITEM_DISTANCE : ALLOWED_DISTANCE ;
ITEM_WEIGHT : ALLOWED_WEIGHT ;
ITEM_COST : PRICES ;
begin
  for JUNK_DIST in DISTANCE_CLASS
  -- Эквивалентная форма этой строки:
  --   for JUNK_DIST in DISTANCE_CLASS'FIRST
  --   .. DISTANCE_CLASS'LAST
  -- или:
  --   for JUNK_DIST in CLOSE_BY ..
  --   LONG_DISTANCE
  -- которая - не такая общая (почему ?)
  loop
    for JUNK_WEIGHT in WEIGHT_CLASS
      loop
        GET(CURR_RATES(JUNK_DIST, JUNK_WEIGHT), 7);
      end loop;
    SKIP_LINE;
  end loop;
  -- Таблица только что проинициализирована. Заметь-
  -- те, что переменные JUNK_DIST и JUNK_WEIGHT не
  -- были явно об'явлены как об'екты типов
  -- DISTANCE_CLASS и WEIGHT_CLASS соответственно.

```

```

-- Вспомните, что параметры цикла (в данном случае
-- - JUNK_DIST и JUNK_WEIGHT) об'являются неявно
-- и принадлежат к тому же типу, что и границы
-- диапазона значений переменной цикла.
GET(ITEM_NO,7);
while ITEM_NO /= 99999999
  loop
    GET(ITEM_WEIGHT,7);
    -- Используется пакет для операций ввода-вывода
    -- с плавающей точкой - WEIGHT_IO.
    GET(ITEM_DISTANCE,7);
    -- Используется пакет для операций ввода-вывода
    -- с плавающей точкой - DISTANCE_IO.
  SKIP_LINE;
  for JUNK_DIST in reverse DISTANCE_CLASS
    loop
      if ITEM_DISTANCE < ACT_DIST_LIM (JUNK_DIST )
        then
          CURR_DIST_CLASS := JUNK_DIST;
        end if;
      end loop;
      for JUNK_WEIGHT in reverse WEIGHT_CLASS
        loop
          if ITEM_WEIGHT < ACT_WEIGHT_LIM (JUNK_WEIGHT)
            then
              CURR_WEIGHT_CLASS := JUNK_WEIGHT;
            end if;
          end loop;
          ITEM_COST :=
            PRICES(CURR_RATES(CURR_DIST_CLASS,
              CURR_WEIGHT_CLASS )
              * PRICES ( ITEM_WEIGHT) );
          NEW_LINE;
          PUT(ITEM_COST,7,2);
          SKIP_LINE;
          GET(ITEM_NO);
        end loop;
      end SHIP_RATE;

```

В следующей программе демонстрируется применение динамических массивов. Программа, названная GRADES, выводит оценки за контрольную работу, которая выполняется студентами путем выбора номеров ответов из представленной совокупности вариантов возможных ответов. Первая строка входного файла содержит целое число в интервале от 20 до 50, равное количеству вопросов в контрольной работе. В следующей строке размещаются ключи ответов.

Например, если в работе 34 вопроса, то во второй строке задается последовательность из 34 правильных номеров вариантов ответов. Каждый номер варианта ответа - это цифра от 1 до 5. Пробелы между цифрами не ставятся. Каждая из остальных строк входного файла содержит наборы ответов студентов.

Личный номер (ID) студента занимает 10 позиций, а сами ответы располагаются с одиннадцатой позиции. Программа должна выводить количество правильных ответов для каждого студента. Признаком конца потока входных данных служит указание в качестве личного номера студента числа 9999999999.



## Программа GRADES

```

with TEXT_IO; use TEXT_IO;
procedure GRADES is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type CHOICES is range 1 .. 5;
  type POSSIBLE_QUESTIONS is range 1 .. 50;
  -- CHOICES и POSSIBLE_QUESTIONS - целые типы.
  package CHO_IO is new INTEGER_IO(CHOICES);
  use CHO_IO;
  package POSS_IO is new
    INTEGER_IO(POSSIBLE_QUESTIONS);
  use POSS_IO;
  NO_QUESTIONS : POSSIBLE_QUESTIONS := 50;
  type ANSWERS is array ( 1 .. NO_QUESTIONS ) of
    CHOICES;
  KEY_ANSWERS, STUDENT_ANSWERS : ANSWERS ;
  GOOD_ANSWERS : INTEGER ;
  STUDENT_ID : STRING ( 1 .. 10);
begin
  GET (NO_QUESTIONS);
  -- Здесь есть два динамических массива -
  -- KEY_ANSWERS и STUDENT_ANSWERS. Количество
  -- элементов в них задается во время выполне-
  -- ния программы, в данном случае после
  -- считывания числа вопросов. Здесь требуется
  -- пакет POSS_IO.
  SKIP_LINE;
  for I in 1 .. NO_QUESTIONS
    loop
      GET ( KEY_ANSWERS (I), 1);
      -- Здесь требуется пакет CHO_IO.
    end loop;
  SKIP_LINE;
  GET ( STUDENT_ID );
  while STUDENT_ID /= "9999999999"
  loop
    GOOD_ANSWERS := 0;
    for J in 1 .. NO_QUESTIONS
      loop
        GET ( STUDENT_ANSWERS(J), 1);
        if STUDENT_ANSWERS(J) = KEY_ANSWERS (J)
          then
            GOOD_ANSWERS := GOOD_ANSWERS + 1 ;
          end if;
        end loop;
      NEW_LINE;
      PUT ( " The number of good answers is " );
      PUT ( GOOD_ANSWERS, 5);
      PUT ( " for the id ");
      PUT ( STUDENT_ID );
      SKIP_LINE;
      GET ( STUDENT_ID ) ;
    end loop;
  end GRADES;

```

## 2.2.4. Регулярные типы с неуточненными границами диапазонов индексов

Неуточненные регулярные типы объявляются по следующей схеме:

`тип имя_типа is array (один_или_более_индексов) of тип_компоненты;`

Часть «один или более индексов» — это последовательность членов вида:

`обозначение_типа range < >`

отделенных друг от друга запятыми. Обозначения типа — это либо имена типов, либо имена подтипов. (О подтипах будет рассказано далее в этой главе.)

Вот некоторые примеры:

`type MATRIX is array (INTEGER range < >, INTEGER range < >) of FLOAT;`

`type VECTOR is array (INTEGER range < >) of FLOAT;`

В качестве объявления предопределенного типа STRING (Строковый) используется следующая конструкция:

`type STRING is array (NATURAL range < >) of CHARACTER;`

где NATURAL (Натуральный) — это предопределенный тип, к которому относятся целые положительные числовые значения. Пара символов < > называется *ромбиком* и обозначает неуточненный диапазон значений индексов для регулярного типа.

Границы индексов у неуточненных массивов следует указывать при объявлении этих объектов. Например:

`Z: MATRIX (1 .. (N - 1)/2, N .. N + 10);`

`Y: MATRIX (-3 .. 3, -2 .. 2);`

`YY: MATRIX (0 .. 7, -1 .. 3);`

Заметьте, что границы индексов у массивов Z и Y в этих объявлениях разные, а типы массивов одинаковые. Вот еще примеры:

`X: STRING (5 .. L ** 2);`

`V: VECTOR (-7 .. 1);`

`VV: VECTOR (-7 .. 2);`

Неуточненные регулярные типы — это весьма гибкое средство, полезное при разработке подпрограмм. Более подробно эти массивы будут описаны в гл. 5.

## 2.2.5. Атрибуты массивов

Далее приведем список некоторых атрибутов, которые можно применять совместно с объектами произвольного регулярного типа A. Если A — уточненный регулярный тип, то эти атрибуты будут применимы также и к нему. Данные атрибуты нельзя употреблять совместно с названиями неуточненных регулярных типов, однако для объектов, принадлежащими к таким типам, эти атрибуты разрешается использовать.

Атрибут	Описание
A'FIRST (A'первый)	Дает нижнюю границу диапазона для первого индекса. Для любого из массивов, принадлежащих к типу SHIPPING_RATES (см. программу SHIP_RATE), значением SHIPPING_RATES'FIRST является CLOSE_BY. Тот же результат получится, если вместо имени данного типа указать имя переменной, относящейся к нему: CURR_RATES'FIRST
A'LAST (A'последний)	Дает верхнюю границу диапазона для первого индекса. Например, SHIPPING_RATES'LAST (см. ту же программу) возвращает результат LONG_DISTANCE

A'LENGTH (A' длина)	Дает количество значений, которое может принимать первый индекс, т.е. размер массива по этому измерению. Для SHIPPING_RATES' LENGTH будет получено число 3
A'RANGE (A' диапазон)	Дает все значения первого индекса, лежащие в диапазоне A'FIRST .. A'LAST. Результатом SHIPPING_RATES'RANGE будут значения из диапазона CLOSE_BY .. LONG_DISTANCE

В многомерных массивах атрибуты можно запросить для любого из индексов. В приведенных ниже атрибутах N обозначает статическое выражение целого типа.

Атрибут	Описание
A'FIRST(N)	Дает нижнюю границу диапазона N-го индекса для регулярного типа A. Например, SHIPPING_RATES'FIRST(2) даст результат LIGHT. Атрибуты A'FIRST и A'FIRST(1) идентичны
A'LAST(N)	Дает верхнюю границу диапазона N-го индекса для регулярного типа A. Например, результатом запроса атрибута SHIPPING_RATES' LAST(2) будет HEAVY. Атрибуты A'LAST и A'LAST(1) идентичны
A'LENGTH(N)	Дает размер массива по N-му измерению. Например, SHIPPING_RATES'LENGTH(2) равно 4. Атрибуты A'LENGTH(1) и A'LENGTH идентичны
A'RANGE(N)	Дает диапазон значений A'FIRST(N) .. A'LAST(N), т.е. фактически подтип. Объяснение будет приведено далее в этой главе. Например, в результате запроса атрибута SHIPPING_RATES' RANGE(2) получится диапазон LIGHT .. HEAVY. Атрибуты A'RANGE(1) и A'RANGE идентичны

## 2.3. ОПЕРАЦИИ С РЕГУЛЯРНЫМИ ТИПАМИ

### 2.3.1. Равенство и неравенство

Операции проверки на равенство и неравенство определены для каждого из двух объектов, принадлежащих к любому из типов, с которыми мы познакомились в гл. 1 и знакомимся в этой главе. Для массивов равенство означает, что все их *согласующиеся компоненты* равны в соответствии с правилами равенства, определенными для того типа, к которому эти компоненты принадлежат. Согласование компонент происходит следующим образом<sup>1)</sup>. При сравнении массивов сначала ставятся в соответствие друг другу нижние границы диапазонов индексов, а затем — последовательно идущие индексные позиции обоих массивов. Если в одном из массивов отсутствует необходимая для согласования компонента, а также если согласующиеся компоненты имеют разные значения, то массивы неравны. Если два массива относятся к одному и тому же типу и компоненты их согласуются, то значение одного массива можно присваивать другому массиву.

**Примеры.** В следующих примерах используются переменные V и VV типа VECTOR, а также Y и YY типа MATRIX (см. предыдущий раздел).

```
V := (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0);
VV := (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0);
```

Здесь V и VV принадлежат к одному и тому же типу, но их компоненты не согласуются, так как в массиве VV больше компонент, чем в V. Поэтому массивы V и VV неравны, а

---

Очевидно, что для «согласованности» компонент необходимо наличие одинакового числа измерений у массивов и одинакового количества компонент по соответствующим измерениям. — Прим. перев.

присваивание невозможно.

```
Y := ((-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0),
      (-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0),
      (-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0),
      (-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0),
      (-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0));
YY := Y;
```

У массивов YY и Y, принадлежащих к одному и тому же типу, компоненты согласуются, и поэтому присваивание разрешается. После присваивания оба массива стали равными. Если затем написать оператор присваивания

```
· YY(0, 3) := 0.0;
```

то Y и YY станут неравны. Обратите внимание на то, как массивы инициализируются с помощью агрегатов.

### 2.3.2. Другие операции отношения

Для регулярных типов допускается и ряд других операций отношения. Они разрешены только для одномерных массивов, компоненты которых принадлежат к дискретным (т.е. к перечисляемым или целым) типам. В указанном случае можно употреблять операции  $<$ ,  $<=$ ,  $>$ ,  $>=$ . Порядок выполнения сравнения при использовании этих операций подчиняется следующему лексикографическому правилу: первая компонента массива имеет наивысший приоритет, она сравнивается первой, затем сравнивается вторая компонента и т.д. Итак, если два одномерных массива принадлежат к одному и тому же типу, а их компоненты относятся к дискретному типу, то при сравнении этих массивов вначале будут сравниваться их первые компоненты. Если они неравны, то большим массивом будет тот, у которого первая компонента больше. Если же первые компоненты равны, то будут сравниваться вторые компоненты и т.д. до тех пор, пока не будет принято решение. Пустой массив не имеет компонент вообще, и поэтому он заведомо меньше массива, имеющего по крайней мере хотя бы одну компоненту.

Вот некоторые примеры:

```
type POSSIBLE_RANGE is range -1000 .. INTEGER'LAST;
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
type EVENTS is array (POSSIBLE_RANGE range <>) of DAY;
LAST_5_IN_NY : EVENTS (1 .. 5) := (MON, TUE, TUE, SAT, FRI);
LAST_3_IN_LA : EVENTS (8 .. 10) := (MON, MON, SAT);
```

**Пример.** Для приведенных выше объявлений можно записать отношение:

```
LAST_5_IN_NY < LAST_3_IN_LA
```

Оно даст логическое значение FALSE, так как вторая компонента массива LAST\_5\_IN\_NY, т.е. компонента LAST\_5\_IN\_NY(2), равная TUE, больше, чем вторая компонента массива LAST\_3\_IN\_LA, т.е. компонента LAST\_3\_IN\_LA(9), которая равна MON.

Однако если написать

```
LAST_3_IN_LA(9) := TUE;
```

то отношение даст результат TRUE, так как первые две компоненты рассматриваемых переменных регулярного типа равны, а LAST\_5\_IN\_NY(3), равная TUE, меньше, чем LAST\_3\_IN\_LA(10), равная SAT.

### 2.3.3. Сцепление

Для одномерных неуточненных массивов существует еще одна операция. Она называется *сцеплением* (catenation) и обозначается символом «&». Пусть массив имеет тип T, а его компоненты – тип C, и действует следующее объявление:

type T is array (INDEX range <>) of C;

Результатом операции сцепления будет массив, индекс первой компоненты которого всегда равен INDEX'FIRST.

Оба операнда могут относиться к типу T, или один из них может быть типа C, а другой – типа T. Результирующий массив содержит компоненты первого операнда, за которыми располагаются компоненты второго операнда. Если один из операндов принадлежит к типу C, то он рассматривается как массив с одной компонентой.

**Пример.** Будет правильной строка

LAST\_5\_IN\_NY & LAST\_3\_IN\_LA

В результате получится одномерный массив, первая компонента которого имеет индекс – 1000. Всего в нем будет 8 компонент. Значения этих компонент можно представить в виде агрегата

(MON,TUE,TUE,SAT,FRI,MON,MON,SAT)

Можно записать

FRI & LAST\_5\_IN\_NY

что даст агрегат

(FRI,MON,TUE,TUE,SAT,FRI)

Индекс первой компоненты будет равен – 1000.

### 2.3.4. Строки

Как упоминалось в предыдущем разделе, тип STRING (Строковый) определяется как одномерный неуточненный регулярный тип:

type STRING is array (NATURAL range <>) of CHARACTER;

Границы индексов задаются во время объявления объектов этого типа.

Вот примеры объявлений строк:

FIELD1 : STRING (1 .. 1 + 7);

FIELD2 : STRING (1 – 7 .. 1 – 1);

FIELD3 : STRING (20 .. 27) := "NEW\_YORK";

FIELD4 : STRING (40 .. 49);

Здесь FIELD3 – строковая переменная, получающая начальное значение при объявлении. Обратите внимание на то, что границы диапазонов индексов у строк FIELD1 и FIELD2 различны, хотя они и принадлежат к одному регулярному типу. Вспомните, что такая ситуация невозможна для уточненных регулярных типов. Дополнительное ограничение для строк заключается в том, что, когда границы диапазона индексов получают некоторые фактические значения (либо при трансляции программы, либо во время ее выполнения), эти фактические значения должны быть положительными целыми числами.

Все атрибуты массивов применимы и к объектам типа STRING, т.е. к строкам, несмотря на то что они не определены для самого типа STRING, так как тип

STRING—это неуточненный регулярный тип. Так, FIELD2'LENGTH даст в результате 7, а FIELD3'FIRST даст 20. Поскольку компоненты для типа STRING относятся к символьному типу, то можно записывать операторы присваивания вида:

FIELD3(21) := 'G';

Эквивалентным способом инициализации переменной FIELD3 будет:

FIELD3 : STRING(20 .. 27) := ('N','E','W','-','Y','O','R','K');

Операции отношения <, <=, >= и >, а также операция сцепления & вполне допустимы для объектов типа STRING, т.е. для строк.

**Пример.** Если

FIELD4 := "CALIFORNIA";

то отношение FIELD4 < FIELD3 даст логическое значение TRUE. Результатом выполнения операции FIELD3& FIELD4 будет строка длиной в 18 символов, индекс первой компоненты которой равен 1.

### 2.3.5. Вырезки

Для одномерных массивов в языке Ада разрешается задавать особое обозначение для последовательности следующих друг за другом компонент, называемых *вырезкой из массива*. К ней можно обращаться. Вырезка—это фактически часть исходного одномерного массива.

**Пример.** Можно обозначить вырезку следующим образом:

V(-5 .. -2)	V—переменная, объявленная ранее как V : VECTOR (-7 .. 1)
V(LL .. RR)	Это обозначение будет корректным, если LL и RR—целые числа, лежащие в требуемом диапазоне (в данном случае от -7 до 1)

Вырезка V(-5 .. -2) содержит четыре компоненты. Первая из них—это V(-5 .. -2) (-5). Она равна V(-5). Четвертая компонента вырезки—это V(-5 .. -2) (-2). Она равна V(-2).

Разрешается присваивание вырезок, что служит полезным средством для компактной записи эквивалентной последовательности операторов присваивания. Например, правильной будет строка

V(-7 .. -4) := V(-5 .. -2);

Этот оператор означает, что вырезка V(-7 .. -4), в которую входят четыре компоненты с индексами от -7 до -4, примет значение вырезки V(-5 .. -2), в которую в свою очередь входят 4 компоненты с индексами от -5 до -2.

Однако при расшифровке смысла этого оператора следует проявить осторожность. «Подводный камень» здесь такой. Вначале выделяются компоненты вырезки V(-5 .. -2)<sup>1)</sup>, и только потом их значения присваиваются соответствующим компонентам вырезки V(-7 .. -4). Поэтому пытаться многократно дублировать значение первой по порядку компоненты вырезки, т.е. повторять присваивание ее значения другим компонентам массива V с помощью оператора

V(-6 .. 1) := V(-7 .. 0);

было бы неправильным. Здесь не произойдет дублирования, так как значение первой компоненты вырезки V(-7 .. 0), т.е. V(-7), присваивается только компоненте V(-6).

<sup>1)</sup> Они записываются в некоторую промежуточную область памяти.—Прим. перев.

Затем вторая компонента вырезки  $V(-7 \dots 0)$ , равная тому значению компоненты  $V(-6)$ , которое имелось до начала выполнения оператора присваивания (а не текущее, уже измененное, значение  $V(-6)$ ), будет помещена в  $V(-5)$  и т. д. Если  $V(-7 \dots 0)$  было равно NEW\_YORK, то после выполнения оператора присваивания значением  $V(-7 \dots 1)$  станет NNEW\_YORK, а вовсе не NNNNNNNNN.

## 2.4. КОМБИНИРОВАННЫЕ ТИПЫ

Массивы представляют собой совокупности компонент, относящихся к одному и тому же типу. Объекты комбинированного типа, которые будем называть *структурами*, — это совокупности поименованных компонент, принадлежащих к различным типам. Объявление комбинированного типа осуществляется в соответствии со схемой:

```

type      имя_комбинированного_типа is
  record
    список_компонент
  end record;
```

Список компонент в простейшей его форме — это список объявлений. Если комбинированный тип не имеет компонент, то на месте этого списка ставится зарезервированное слово null (пусто). Объявления комбинированных типов могут также содержать *дискриминантную* и/или *вариантную* части. Структуры с дискриминантами будут рассмотрены далее в этом разделе, а структуры с вариантами — в гл. 4.

### 2.4.1. Особенности комбинированных типов

Вот пример объявления комбинированного типа:

```

type EMPLOYEE is
  record
    FIRST_NAME      : STRING (1 .. 20);
    LAST_NAME       : STRING (1 .. 20);
    HOME_ADDRESS    : STRING (1 .. 30);
    SOC_SEC_NO      : STRING (1 .. 9);
    HOURLY_RATE     : FLOAT ;
    HOURS_WORKED    : INTEGER range 1 .. 168;
  end record;
```

Здесь имя\_комбинированного\_типа — это EMPLOYEE, а компоненты — это переменные, перечисленные после зарезервированного слова record, начиная с FIRST\_NAME и кончая HOURS\_WORKED.

Вот еще объявления комбинированных типов:

```

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
type DAY_NO is 1 .. 31;
type MONTH is (JANUARY, FEBRUARY, MARCH, APRIL, MAY,
               JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER,
               NOVEMBER, DECEMBER);
type YEAR is 1900 .. 2050 ;
type DATE is
  record
    WEEK_DAY : DAY;
    MONTH_NAME : MONTH;
    DAY_NO : DAY;
    YEAR_NO : YEAR;
  end record;
```

А вот — объявления структур, т.е. объектов комбинированного типа:

```
CURR_EMPLOYEE, PREV_EMPLOYEE : EMPLOYEE;
ACTUAL_DATE, SETTLEMENT_DATE : DATE;
```

Обратиться к компоненте структуры можно с помощью селектора (selected-component notation). При этом вначале ставится имя объекта комбинированного типа (префикс), а затем точка и имя компоненты (селектор).

**Пример.** Строка

```
ACTUAL_DATE.WEEK_DAY
```

означает обращение к компоненте WEEK\_DAY переменной ACTUAL\_DATE, принадлежащей к комбинированному типу DATE. Сама компонента относится к типу DAY. Строка

```
CURR_EMPLOYEE.HOME_ADDRESS
```

позволяет выбрать компоненту HOME\_ADDRESS переменной CURR\_EMPLOYEE типа EMPLOYEE.

Для структур, как и для массивов, можно строить агрегаты, которые позволяют выполнять присваивание некоторого значения объектам комбинированного типа. Здесь имеются в виду так называемые позиционные агрегаты-структуры. Позиционный агрегат-структура состоит из заключенного в скобки полного набора значений, разделенных запятыми. Например, можно сделать следующее объявление:

```
STARTING_DATE : constant DATE := (MON,AUGUST,20,1984);
```

Здесь объявлена константа STARTING\_DATE, ее значением является агрегат (MON,AUGUST, 20,1984).

Объекты одного и того же комбинированного типа, не имеющего ни дискриминантной, ни вариантной частей, имеют идентичную совокупность названий компонент.

Две структуры одного типа можно сравнивать на равенство или неравенство. Две структуры считаются равными, если все одноименные компоненты, входящие в них, равны. В противном случае структуры неравны. Однако две пустые структуры одного и того же типа всегда считаются равными. Для комбинированных типов операции отношения <, <=, >= и > не определены.

Для структур одного и того же типа разрешено присваивание. Оно означает, что значение каждой компоненты структуры, расположенной справа от символа присваивания, замещает значение одноименной компоненты структуры, расположенной слева. Например, оператор

```
ACTUAL_DATE := SETTLEMENT_DATE;
```

является корректным оператором присваивания. Он означает, что

```
ACTUAL_DATE.WEEK_DAY      :=SETTLEMENT_DATE.WEEK.DAY;
ACTUAL_DATE.MONTH_NAME    :=SETTLEMENT_DATE.MONTH.NAME;
ACTUAL_DATE.DAY_NO        :=SETTLEMENT_DATE.DAY.NO;
ACTUAL_DATE.YEAR_NO       :=SETTLEMENT_DATE.YEAR.NO;
```

Компоненты, задаваемые в объявлении комбинированного типа, могут в свою очередь принадлежать к комбинированному или регулярному типу. Например, можно объявить такой комбинированный тип:



```

type MORE_INFO is
  record
    NEW_EMPLOYEE : EMPLOYEE;
    DATE_HIRED   : DATE;
  end record;

```

Тогда при наличии объявления переменной

```
EMPL : MORE_INFO;
```

конструкция `EMPL..DATE_HIRED..DAY_NO` означает выборку компоненты `DAY_NO` структуры `DATE_HIRED`, которая в свою очередь служит компонентой структуры — переменной `EMPL` типа `MORE_INFO`. В языке Ада отсутствуют предопределенные комбинированные типы.

В данном разделе мы не будем обсуждать атрибуты, применимые к комбинированным типам. Отметим только, что существуют атрибуты, которые вырабатывают значения относительных смещений компонент, адресов первого и последнего бита компоненты и некоторые другие, машинно-зависимые, атрибуты.

## 2.4.2. Программа, в которой используются комбинированные типы

В программе, представленной ниже, применяются комбинированные типы. Программа выводит имя служащего, принятого на работу последним. Входные данные программы размещаются в строках, каждая из которых имеет следующие поля: `LAST_NAME` (фамилия) — поз. 1–20; `YEAR_NO` (год) — поз. 21–22; `MONTH_NO` (месяц) — поз. 23–24; `DAY_NO` (число) — поз. 25–26. Последние три поля содержат дату приема на работу. Признаком конца потока входных данных служит строка, в поле `LAST_NAME` которой записано: 12345678901234567890.

### Программа LAST\_HIRED

```

with TEXT_IO ; use TEXT_IO;
procedure LAST_HIRED is
  type DATE is
    record
      YEAR_NO : INTEGER range 50 .. 99;
      MONTH_NO: INTEGER range 1 .. 12;
      DAY_NO  : INTEGER range 1 .. 31;
    end record ;
  type EMPLOYEE is
    record
      LAST_NAME : STRING(1 .. 20);
      HIRING_DATE: DATE;
    end record;
  CURR_EMPL, LAST_EMPL : EMPLOYEE ;
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  begin
    LAST_EMPL := ("JUST AN EARLY DATE ", (50, 1, 1));
    -- Здесь выполняется инициализация самой ранней
    -- датой.
    GET (CURR_EMPL.LAST_NAME);
    while CURR_EMPL.LAST_NAME /=
      "12345678901234567890"
    loop
      GET (CURR_EMPL.HIRING_DATE.YEAR_NO, 2);

```

```

GET (CURR_EMPL.HIRING_DATE.MONTH_NO,2);
GET (CURR_EMPL.HIRING_DATE.DAY_NO,2);
if CURR_EMPL.HIRING_DATE.YEAR_NO >
    LAST_EMPL.HIRING_DATE.YEAR_NO      or
    CURR_EMPL.HIRING_DATE.YEAR_NO      =
    LAST_EMPL.HIRING_DATE.YEAR_NO      and
    ( CURR_EMPL.HIRING_DATE.MONTH_NO   >
    LAST_EMPL.HIRING_DATE.MONTH_NO     or
    CURR_EMPL.HIRING_DATE.MONTH_NO     =
    LAST_EMPL.HIRING_DATE.MONTH_NO     and
    CURR_EMPL.HIRING_DATE.DAY_NO       =
    LAST_EMPL.HIRING_DATE.DAY_NO )
then
    LAST_EMPL := CURR_EMPL ;
end if;
SKIP_LINE;
GET(CURR_EMPL.LAST_NAME);
end loop;
NEW_LINE; PUT(" The last hired is ");
PUT(LAST_EMPL.LAST_NAME);
end LAST_HIRED;

```

На рис. 2.3 показаны значения объектов, объявленных в программе LAST\_HIRED в момент времени после считывания последней строки данных. Тестовые данные приведены на том же рисунке.

### 2.4.3. Программа, демонстрирующая использование введенных понятий

Рассмотрим еще одну программу на Аде, в которой используются многие из понятий, введенных в первых двух главах. В частности, в ней употребляются комбинированные типы.

В каждой строке входных данных содержатся сведения о лицах, работающих по контракту. Формат данных таков:

Позиции	Данные
1-9	Личный номер служащего
10-30	Фамилия служащего
31-36	Дата начала работы в формате ГГММДД, например 840431
37-42	Дата конца контракта в формате ГГММДД
43-48	Дневной заработок (в долларах и центах)

#### Входные данные

```

STEVEN K. KNIGHT  820317
B. B. HANSEN      820425
LOU HARRIS        820901
12345678901234567890

```

#### Значения объектов

CURR\_EMPL

LAST\_EMPL

CURR\_EMPL.LAST\_NAME    CURR\_EMPL.HIRING\_DATE

Lou Harris	820901
------------	--------

LAST\_EMPL.LAST\_NAME    LAST\_EMPL.HIRING\_DATE

STEVEN K. KNIGHT	820317
------------------	--------

Рис. 2.3. Входные данные и значения объектов для программы LAST\_HIRED.

Служащим, работающим по контракту, начисляется некоторая ежедневная заработная плата. За работу по субботам оплата удваивается, а за работу в воскресенье утраивается.

Программа должна для каждого служащего выводить его фамилию, личный номер, количество отработанных дней, причитающуюся сумму заработка и дату выдачи заработной платы, которая производится через пять дней после окончания работы. Дата выдачи зарплаты имеет формат:

день\_недели, месяц число, год

Например, может быть такая дата: Tuesday, August 21, 1984 (21 августа 1984 г., вторник). В последней строке входных данных указывается личный номер, равный 999999999. Программа проверяет правильность каждой даты. Например, недопустимо 31-е число в апреле.

### Программа DATE\_CONVERSION

```
with TEXT_IO; use TEXT_IO;
procedure DATE_CONVERSION is
  type DAY is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY);
  type DAY_INT is range 1 .. 31;
  type JULIAN_DAYS is range 1 .. 366 ;
  type MONTH is (JANUARY, FEBRUARY, MARCH, APRIL, MAY,
    JUNE, JULY, AUGUST, SEPTEMBER,
    OCTOBER, NOVEMBER, DECEMBER);
  type YEAR is range 00 .. 2050 ;
  type MONTH_INT is range 1 .. 12;
  type DATE is
    record
      WEEK_DAY : DAY;
      MONTH_NAME : MONTH;
      MONTH_NO : MONTH_INT;
      DAY_NO : DAY_INT;
      YTD_DAYS : JULIAN_DAYS;
      TOTAL_DAYS : INTEGER;
      YEAR_NO : YEAR;
    end record;
  BASE_DATE : constant DATE :=
    (MONDAY, JANUARY, 1, 1, 1, 1, 1984);
  -- В данной программе предполагается, что на
  -- вход подаются только даты, располагающиеся
  -- после 1 января 1984г. Для других дат из-
  -- мените должным образом эту константу.
  BASE_LEAP : constant INTEGER :=
    INTEGER(BASE_DATE.YEAR_NO) / 4 +
    INTEGER(BASE_DATE.YEAR_NO) / 400 -
    INTEGER(BASE_DATE.YEAR_NO) / 100;
  -- Эта константа равна количеству високосных
  -- лет, прошедших от нулевого года до года
  -- BASE_DATE.YEAR_NO. Обратите внимание, что
  -- при ее вычислении используется статичес-
  -- кое выражение и преобразование типа.
  INPUT_DATE, HOLD_DATE : DATE;
  NO_OF_SATURDAYS, NO_OF_SUNDAYS : INTEGER;
  type CONTRACTOR is
    record
```

```

ID_NO : STRING ( 1 .. 9 ) ;
CO_NAME: STRING ( 1 .. 21 ) ;
end record;
type INFO_LINE is
record
    CURR_CONTRACTOR : CONTRACTOR ;
    DAYS_WORKED      : INTEGER ;
    DUE_DATE         : DATE ;
end record;
PROC_LINE : INFO_LINE;
type DAYS_IN_MONTH is array (MONTH_INT,BOOLEAN)
                                of DAY_INT;
ACTUAL_DAYS_IN_YEAR : constant DAYS_IN_MONTH :=
( (31,31),(28,29),(31,31),(30,30),
  (31,31),(30,30),(31,31),(31,31),
  (30,30),(31,31),(30,30),(31,31) );
-- Мы объявили массив-константу, индексируемый
-- значениями типа MONTH_INT в диапазоне от 1
-- до 12 и логическими значениями - всего
-- лишь двумя (FALSE и TRUE). FALSE означает,
-- что представленный год - не високосный.
type FEE is delta 0.0001 range 0.000 ..
                                500_000.000 ;
-- Объявлен фиксированный тип.
-- Вам следует удостовериться, разрешено
-- ли в вашей версии Ады такое количество
-- цифр в этом числе.
package FEE_IO is new FIXED_IO(FEE);
use FEE_IO;
CONTR_DAILY_FEE, CONTR_TOTAL_FEE : FEE;
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
GOOD_DAY, GOOD_YEAR, LEAP_YEAR : BOOLEAN;
package DAY_IO is new ENUMERATION_IO(DAY);
use DAY_IO;
package MONTH_IO is new ENUMERATION_IO(MONTH);
use MONTH_IO;
package YEAR_IO is new INTEGER_IO(YEAR);
use YEAR_IO;
package MONTH_INT is new INTEGER_IO(MONTH_INT);
use MONTH_INT;
package DAY_INT_IO is new INTEGER_IO(DAY_INT);
use DAY_INT_IO;
INPUT_LEAP : INTEGER ;
XTRA_DAYS  : INTEGER ;
procedure COUNT_DAYS_AND_CHECK is
-- Эта процедура заполнит оставшиеся компоненты
-- структуры INPUT_DATE. В ней уже присвоены
-- значения компонентам DAY_NO, MONTH_NO и
-- YEAR_NO. Вспомните, что в соответствии с ма-
-- териалом разд.1.5 объекты, объявленные в про-
-- цедуре DATE_CONVERSION, будут известны, и в
-- данной процедуре их можно использовать.
begin
-- Правильно ли задан год? Если нет, то
-- GOOD_YEAR := FALSE;
if INPUT_DATE.YEAR_NO < 1900
then

```

```

INPUT_DATE.YEAR_NO := INPUT_DATE.YEAR_NO + 1900;
end if;
if INPUT_DATE.YEAR_NO < BASE_DATE.YEAR_NO
then
GOOD_YEAR := FALSE;
else
GOOD_YEAR := TRUE;
end if;
-- Мы не проверяем номер месяца. Если он выходит
-- за интервал 1 .. 12, то при присвоении номера
-- месяца возникнет исключительная ситуация, сви-
-- детельствующая об ошибке.
-- Число - правильное? Если нет, то
-- GOOD_DAY := FALSE; Но сперва выясним, не висо-
-- косный ли это год. См также программу
-- DAY_CONVERSION из разд.1.4.
if INPUT_DATE.YEAR_NO rem 4 = 0 and
INPUT_DATE.YEAR_NO rem 100 /= 0 or
INPUT_DATE.YEAR_NO rem 400 = 0
then
LEAP_YEAR := TRUE;
else
LEAP_YEAR := FALSE;
end if;
-- Теперь можно употребить LEAP_YEAR для обраче-
-- ния к нужной строке массива
-- ACTUAL_DAYS_IN_YEAR.
if INPUT_DATE.DAY_NO >
ACTUAL_DAYS_IN_YEAR( INPUT_DATE.MONTH_NO,
LEAP_YEAR)
then
GOOD_DAY := FALSE;
else
GOOD_DAY := TRUE;
end if;
-- Теперь найдем YTD_DAYS для этой даты.
if GOOD_DAY
then
INPUT_DATE.YTD_DAYS :=
JULIAN_DAYS(INPUT_DATE.DAY_NO);
-- Здесь необходимо преобразование типа.
if INPUT_DATE.MONTH_NO > 1
then
for I in 1 .. INPUT_DATE.MONTH_NO - 1
loop
INPUT_DATE.YTD_DAYS := INPUT_DATE.YTD_DAYS +
JULIAN_DAYS(ACTUAL_DAYS_IN_YEAR
(I, LEAP_YEAR));
end loop;
end if;
end if;
-- Найдем, сколько дней прошло от даты отсчета
-- до представленной даты. Вначале выясним,
-- сколько прошло високосных лет до введенной
-- даты.
INPUT_LEAP := INTEGER(INPUT_DATE.YEAR_NO) / 4 +
INTEGER(INPUT_DATE.YEAR_NO) / 400 -
INTEGER(INPUT_DATE.YEAR_NO) / 100 ;

```

```

INPUT_DATE.TOTAL_DAYS := 365 * INTEGER((
    INPUT_DATE.YEAR_NO - BASE_DATE.YEAR_NO )) +
    INTEGER(INPUT_DATE.YTD_DAYS) + INPUT_LEAP -
    BASE_LEAP;
-- Найдем день недели и название месяца.
INPUT_DATE.MONTH_NAME :=
    MONTH'VAL(INPUT_DATE.MONTH_NO);
-- Вспомните, что атрибут VAL применим для
-- перечисляемых типов.
INPUT_DATE.WEEK_DAY := DAY'VAL (
    (INPUT_DATE.TOTAL_DAYS + DAY'POS(
        BASE_DATE.WEEK_DAY) - 2) mod 7 + 1 );
-- Это выражение - довольно сложное.
-- Вначале вычисляется позиция дня недели
-- для даты отсчета BASE_DATE. Позиция
-- равна, например, 2 для вторника
-- (TUESDAY). Она добавляется к общему ко-
-- личеству дней минус 2. Остаток от деле-
-- ния этого числа на 7 плюс 1 дает отно-
-- сительную позицию представленного дня
-- недели (целое число в интервале от 1 до
-- 7). Вы сможете лучше понять смысл этого
-- выражения, если попытаетесь вычислить
-- его вручную для нескольких дат.
end COUNT_DAYS_AND_CHECK;
begin
GET(PROC_LINE.CURR_CONTRACTOR.ID_NO);
while PROC_LINE.CURR_CONTRACTOR.ID_NO /= "999999999"
loop
GET(PROC_LINE.CURR_CONTRACTOR.CO_NAME);
GET(INPUT_DATE.YEAR_NO,2);
GET(INPUT_DATE.MONTH_NO,2);
GET(INPUT_DATE.DAY_NO,2);
-- Начальная дата прочитана в формате ГГММДД.
-- Обратите внимание, что для этих трех опера-
-- торов GET понадобились три различных пакета.
COUNT_DAYS_AND_CHECK;
-- Вызов процедуры COUNT_DAYS_AND_CHECK. Обра-
-- ботка конечной даты будет выполняться только
-- в том случае, если начальная дата задана
-- корректно.
if GOOD_YEAR and GOOD_DAY
then
HOLD_DATE := INPUT_DATE;
-- Запоминается начальная дата. Здесь исполь-
-- зуется присваивание структур.
GET(INPUT_DATE.YEAR_NO,2);
GET(INPUT_DATE.MONTH_NO,2);
GET(INPUT_DATE.DAY_NO,2);
COUNT_DAYS_AND_CHECK;
if GOOD_YEAR and GOOD_DAY
then
-- Вычисляется количество отработанных дней.
PROC_LINE.DAYS_WORKED :=
    INPUT_DATE.TOTAL_DAYS -
    HOLD_DATE.TOTAL_DAYS + 1;
GET(CONTR_DAILY_FEE,6);
-- Определим, сколько было суббот.

```

```

NO_OF_SATURDAYS := PROC_LINE.DAYS_WORKED / 7;
if DAY_POS(HOLD_DATE.WEEK_DAY) +
  INFO_LINE.DAYS_WORKED mod 7 - 1 > 5
  then
    NO_OF_SATURDAYS := NO_OF_SATURDAYS + 1;
  end if;
-- Определим, сколько было воскресений.
NO_OF_SUNDAYS := PROC_LINE.DAYS_WORKED / 7;
if DAY_POS(HOLD_DATE.WEEK_DAY) +
  PROC_LINE.DAYS_WORKED mod 7 - 1 > 6
  then
    NO_OF_SUNDAYS := NO_OF_SUNDAYS + 1;
  end if;
-- Теперь выясним дату выдачи зарплаты. Это
-- важно для банков. Пять рабочих дней на прак-
-- тике означают неделю срока - возможно, на
-- один или два дня меньше, если дата окончания
-- договора выпадет на выходные дни.
if INPUT_DATE.WEEK_DAY = SATURDAY
  then
    PROC_LINE.DUE_DATE.TOTAL_DAYS :=
      INPUT_DATE.TOTAL_DAYS + 6;
    PROC_LINE.DUE_DATE.WEEK_DAY := FRIDAY;
  elsif INPUT_DATE.WEEK_DAY = SUNDAY
  then
    PROC_LINE.DUE_DATE.TOTAL_DAYS :=
      INPUT_DATE.TOTAL_DAYS + 5;
    PROC_LINE.DUE_DATE.WEEK_DAY := FRIDAY;
  else
    PROC_LINE.DUE_DATE.TOTAL_DAYS :=
      INPUT_DATE.TOTAL_DAYS + 7;
    PROC_LINE.DUE_DATE.WEEK_DAY :=
      INPUT_DATE.WEEK_DAY;
  end if;
XTRA_DAYS := PROC_LINE.DUE_DATE.TOTAL_DAYS -
  INPUT_DATE.TOTAL_DAYS;
-- Теперь выясним, попадает ли дата выплаты
-- на новый месяц или на новый год.
if INPUT_DATE.DAY_NO + DAY_INT(XTRA_DAYS) <
  ACTUAL_DAYS_IN_YEAR (INPUT_DATE.MONTH_NO,
    LEAP_YEAR)
  then
    PROC_LINE.DUE_DATE.DAY_NO :=
      INPUT_DATE.DAY_NO + XTRA_DAYS;
    PROC_LINE.DUE_DATE.MONTH_NAME :=
      INPUT_DATE.MONTH_NAME;
    PROC_LINE.DUE_DATE.YEAR_NO :=
      INPUT_DATE.YEAR_NO;
  else
    PROC_LINE.DUE_DATE.DAY_NO :=
      INPUT_DATE.DAY_NO + DAY_INT(XTRA_DAYS) -
      ACTUAL_DAYS_IN_YEAR (INPUT_DATE.MONTH_NO,
        LEAP_YEAR);
    if INPUT_DATE.MONTH_NAME < DECEMBER
      then
        PROC_LINE.DUE_DATE.MONTH_NAME :=
          MONTH_SUCC(INPUT_DATE.MONTH_NAME);

```

```

PROC_LINE.DUE_DATE.YEAR_NO :=
    INPUT_DATE.YEAR_NO;
else
    PROC_LINE.DUE_DATE.MONTH_NAME := JANUARY;
    PROC_LINE.DUE_DATE.YEAR_NO :=
        INPUT_DATE.YEAR_NO + 1;
end if;
end if;
-- Вычислить сумму причитающейся зарплаты.
CONTR_TOTAL_FEE := FEE( CONTR_DAILY_FEE *
    FEE ( PROC_LINE.DAYS_WORKED *
        (1 + NO_OF_SATURDAYS + 2*NO_OF_SUNDAYS));
-- Вспомните, что результат умножения
-- объектов фиксированного типа FEE уже
-- не относится к типу FEE.
-- В заключение напечатаем результаты.
PUT(PROC_LINE.CURR_CONTRACTOR.ID_NO);
PUT(PROC_LINE.CURR_CONTRACTOR.CO_NAME);
PUT(" Days worked " );
PUT( PROC_LINE.DAYS_WORKED,5);
NEW_LINE;
PUT(" Amount due :");
PUT(CONTR_TOTAL_FEE,10,2);
PUT(" Due date :");
PUT(PROC_LINE.DUE_DATE.WEEK_DAY);PUT(",");
PUT(PROC_LINE.DUE_DATE.MONTH_NAME);
PUT(PROC_LINE.DUE_DATE.DAY_NO,5);PUT(",");
PUT(PROC_LINE.DUE_DATE.YEAR_NO,5);
else
    PUT(" Bad ending period ");
end if;
else
    PUT(" Bad starting period ");
end if;
SKIP_LINE;
GET(PROC_LINE.CURR_CONTRACTOR.ID_NO);
end loop;
end DATE_CONVERSION;

```

#### 2.4.4. Дискриминанты комбинированных типов

В языке Ада имеется средство, называемое дискриминантом, которое позволяет определить семейство комбинированных типов. *Дискриминант* — это именованная компонента каждого объекта данного комбинированного типа. Ее имя должно появиться перед именами компонент, перечисленных в определении этого комбинированного типа.

Объявление комбинированного типа с дискриминантами имеет форму:

```

type имя_комбинированного_типа
    (объявления_дискриминантов) is
    record
        список_компонент
    end record;

```



Объявления дискриминантов<sup>1)</sup> заканчиваются символом ";". Каждое такое объявление очень похоже на обычные объявления объектов дискретного типа. Дискриминантная часть — это последовательность объявлений дискриминантов, заключенная в скобки.

**Примеры.** Пример объявления комбинированного типа с дискриминантами:

```
type PAGES is array (NATURAL range <>, NATURAL range <>) of CHARACTER;
```

Тип NATURAL (Натуральный) охватывает положительные целые числа. Как будет видно из материала следующего раздела, это — предопределенный подтип.

Другой пример:

```
type LINE_NUMBERS is array (NATURAL range <>) of INTEGER;
```

Выше были объявлены два неуточненных регулярных типа, объекты которых используются далее.

```
type PAGE_FORMAT ( NO_OF_LINES : POSITIVE := 55 ;
                   NO_OF_COLUMNS : POSITIVE := 80 );
is
record
  STD_HEADER : STRING ( 1 .. NO_OF_COLUMNS );
  STD_BODY   : PAGES   ( 1 .. NO_OF_LINES,
                        1 .. NO_OF_COLUMNS );
  STD_LINES  : LINE_NUMBERS ( 1 .. NO_OF_LINES );
  STD_FOOTER : STRING ( 1 .. NO_OF_COLUMNS );
end record;
```

Тип PAGE\_FORMAT — комбинированный тип с дискриминантами. Дискриминантная часть содержит два объявления дискриминантов. Первое из них — это

```
NO_OF_LINES : POSITIVE := 55
```

После него стоит символ «;». Объявление второго дискриминанта:

```
NO_OF_COLUMNS : POSITIVE := 80
```

Оба объявления задают начальные значения, которые дискриминанты могут принимать по умолчанию. Начальные значения либо должны быть указаны для всех дискриминантов из дискриминантной части объявления комбинированного типа, либо не указываются вовсе.

Внутри объявления комбинированного типа имена дискриминантов (в нашем примере — это NO\_OF\_COLUMNS и NO\_OF\_LINES) можно использовать в качестве границ диапазонов индексов. В данном примере значения границ задаются с помощью NO\_OF\_COLUMNS для компонент STD\_HEADER, STD\_BODY и STD\_FOOTER. При помощи идентификатора NO\_OF\_LINES указываются границы индексов для компонент STD\_BODY и STD\_LINES.

Имя дискриминанта можно использовать только самостоятельно, т.е. оно не должно входить в состав более сложных выражений. Например, было бы неправильным такое объявление компоненты, входящей в тип PAGE\_FORMAT :

```
STD_COLUMN : STRING ( 1 .. NO_OF_COLUMNS - 1 );
```

Имя дискриминанта также можно употреблять в вариантных частях (см. гл. 4). Его можно использовать и для указания значения другого дискриминанта при объявлении иного (вложенного) комбинированного типа со своим дискриминантом.

Применение дискриминанта в типе PAGE\_FORMAT дало возможность определять целое семейство типов PAGE\_FORMAT. Фактически же значения величин

<sup>1)</sup> Кроме последнего. — Прим. перев.

NO\_OF\_LINES и NO\_OF\_COLUMNS задаются во время объявления объектов, принадлежащих к этому типу. Примером такого объявления структуры может служить:

```
TITLE_PAGE : PAGE_FORMAT (25, 45);
```

Здесь объявляется объект типа PAGE\_FORMAT, у которого NO\_OF\_LINES равно 25, а NO\_OF\_COLUMNS равно 45. А вот другие объявления объектов:

```
REGULAR_PAGE : PAGE_FORMAT(45,70);
MARGIN_WIDTH : constant INTEGER := 10;
APPENDIX_PAGE, INDEX_PAGE :
    PAGE_FORMAT (40, MARGIN_WIDTH + 30 );
```

Фактические значения дискриминантов, задаваемые при объявлении структур, называются *спецификациями дискриминантов*. Полный список спецификаций дискриминантов, разделенных запятыми и заключенных в скобки, называется *уточнением дискриминантов*. Для структуры INDEX\_PAGE, объявленной выше, уточнение дискриминантов — это

(40, MARGIN\_WIDTH + 30)

Здесь — две спецификации дискриминантов. Первая — 40, а вторая — MARGIN\_WIDTH + 30. Как видно из этого примера, спецификации дискриминантов могут быть выражениями. Начиная с гл. 4, в нескольких программах будут широко употребляться структуры с дискриминантами.

Рис. 2.4. иллюстрирует объявления структур, принадлежащих к комбинированному типу PAGE\_FORMAT.

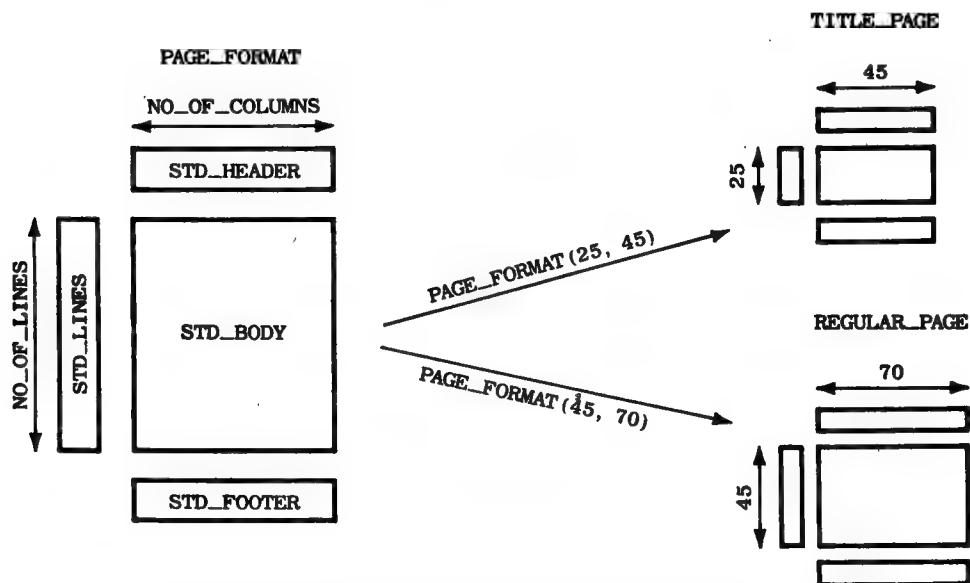


Рис. 2.4. Комбинированные типы с дискриминантами в программе PAGE\_FORMAT.

Если при объявлении типа задаются начальные значения дискриминантов, то указывать при объявлении объектов данного типа уточнение дискриминантов не обязательно, так как дискриминанты при отсутствии уточнения примут по умолчанию эти начальные значения. Например, будет правильным такое объявление объекта:

```
ANY_PAGE : PAGE_FORMAT;
```

Здесь по умолчанию будут приняты значения дискриминантов (55, 80). Если потребуются другие значения, то следует задавать соответствующее уточнение дискриминантов при каждом объявлении объектов.

Дискриминантам нельзя непосредственно присваивать значения. Например, было бы ошибкой написать

```
ANY_PAGE.NO_OF_LINE := 2;
```

Однако можно изменить значения дискриминантов, если присвоить значения сразу всем компонентам структуры, например:

ANY_PAGE := (2, 2,	NO_OF_LINES и NO_OF_COLUMNS получают значение 2
"AA",	"AA" присваивается компоненте STD_HEADER
(('X', 'Y'), ('M', 'N')),	STD_BODY получает значения 'X', 'Y' для первого ряда и 'M', 'N' для второго
(1, 2),	STD_LINES получает значения 1, 2
"VV");	"VV" присваивается компоненте STD_FOOTER

Во всех остальных отношениях дискриминанты ведут себя как обычные компоненты структуры. Следует также отметить, что если присутствует уточнение дискриминант, то значения дискриминантов изменять больше нельзя. Поэтому оператор присваивания, который был правильным для структуры ANY\_PAGE, так как в ней отсутствовали уточнения дискриминантов, будет ошибочным, скажем, для структуры TITLE\_PAGE, поскольку в ней уточнения заданы при объявлении.

Как видно из приведенного выше примера присваивания агрегата структуре ANY\_PAGE, во многих ситуациях позиционная форма агрегатов трудно воспринимается при чтении программы. В гл. 4 будет представлена иная, более удобная для восприятия форма агрегата.

Объекты одного и того же комбинированного типа с дискриминантами можно сравнивать друг с другом на равенство/неравенство. В этом случае будет проверяться равенство или неравенство каждой из соответствующих компонент этих структур. Так, для приведенных выше объявлений структура REGULAR\_PAGE не может быть равной структуре INDEX\_PAGE, если в REGULAR\_PAGE.STD\_BODY будет иное число строк и/или столбцов, чем в INDEX\_PAGE.STD\_BODY.

## 2.5. ПОДТИПЫ И ПРОИЗВОДНЫЕ ТИПЫ

Вспомните, что типы определяют совокупности значений и операции, разрешенные к использованию с этими значениями. К настоящему моменту мы познакомились со следующими типами языка Ада: перечисляемыми, целыми, действительными, регулярными и комбинированными. В языке Ада есть еще два типа: *ссылочные типы* (см. следующую главу) и *приватные типы* (см. гл. 7). Кроме того, в Аде существуют *подтипы* и *производные типы*, которые и освещаются в данном разделе.

### 2.5.1. Подтипы

Программист может пожелать ограничить диапазон возможных значений для типа, но сохранить при этом имевшийся набор операций. Этого можно достичь путем определения так называемого *подтипа* для заданного типа. Ограничение диапазона значений называется *уточнением*, а исходный тип называется *базовым типом*. Тип можно рассматривать и как свой собственный подтип, а поэтому тип является своим собственным базовым типом. В языке Ада есть четыре вида уточнений: уточнение диапазона значений, указание точности, уточнение диапазона индексов и уточнение

дискриминанта. Их примеры были приведены в двух первых главах книги при объявлениях различных типов. Примеры из данного раздела продемонстрируют применение уточнений Ады для подтипов этого языка.

Форма объявления подтипа следующая:

```
subtype имя_подтипа is обозначение_типа;
```

Если присутствует какое-либо уточнение, то объявление принимает вид

```
subtype имя_подтипа is обозначение_типа уточнение;
```

Обозначение\_типа — это либо имя\_типа, либо другое имя\_подтипа. Уточнение относится к одному из четырех перечисленных выше.

**Пример.** Объявления подтипа с уточнением диапазона значений может быть таким:

```
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

Этот предопределенный подтип типа INTEGER (Целый) охватывает значения от 1 до наибольшего целого числа, разрешаемого конкретной реализацией языка. Все операции, допустимые для типа INTEGER, будут разрешены и для подтипа POSITIVE. Здесь присутствует уточнение диапазона значений. Если, скажем, при вычитании одного положительного числа из другого получится отрицательное число, то возникнет исключительная ситуация «нарушение уточнения» (constraint error).

Границы диапазона значений (range constraint) в определении подтипа могут быть, как показывает следующий пример, и статическими, и динамическими выражениями.

```
subtype MARGIN is INTEGER range LEFT_MARGIN + 5 .. RIGHT_MARGIN;
```

Здесь левая граница LEFT\_MARGIN + 5 — это либо статическое, либо динамическое выражение, значение которого не должно превосходить правой границы во время выполнения программы.

**Пример.** Для типа

```
type DAY is (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
```

можно определить подтипы:

```
subtype WEEKEND is DAY range SATURDAY .. SUNDAY;
subtype WORKDAY is DAY range MONDAY .. FRIDAY;
CURR_DAY : WORKDAY;
```

Эти типы являются подтипами перечисляемого типа DAY. Одни и те же атрибуты применимы как к типу DAY, так и к его подтипам. Если переменным присваиваются значения, выходящие за границы указанного диапазона, то возникает исключительная ситуация «нарушение уточнения».

Это произойдет, например, при выполнении оператора

```
CURR_DAY := SATURDAY;
```

Указания точности (accuracy constraints) применяются для действительных типов, т. е. и для плавающих, и для фиксированных. Вот примеры для плавающих типов:

```
type HIGH_PRECISION is digits 15;
subtype LESS_PRECISION is HIGH_PRECISION digits 8;
subtype TIGHT_RANGE is HIGH_PRECISION range - 2.00 .. 2.00;
```

Значения, принадлежащие к подтипу LESS\_PRECISION, — это значения типа HIGH\_PRECISION, но только с восемью значащими цифрами. Количество цифр, задаваемое при определении подтипа, должно быть целым положительным числом, меньшим или равным количеству цифр в базовом типе. Поэтому неверной была бы строка:

subtype BAD\_FLOAT is HIGH\_PRECISION digits 16;

но можно записать

subtype GOOD\_FLOAT is HIGH\_PRECISION digits 14;

Вот некоторые примеры указаний точности в объявлениях фиксированных подтипов.

```
type FEE is delta 0.0001 range 0.00 .. 500_000.00;
sybtype CHECK is FEE delta 0.005
                      range 0.01 .. 5000.00;
sybtype PAYOFF is FEE delta 0.002;
sybtype SMALL_FEE is FEE range 0.01 .. 5000.00;
sybtype PETTY_CASH is SMALL_FEE range 0.01 ..
                                     200.00;
```

Обратите внимание на то, что значение абсолютной точности (delta) для подтипа не может быть меньшим, чем значение точности у исходного типа. Заметьте также, что для фиксированного подтипа уточнение диапазона значений может отсутствовать, при этом берется весь диапазон значений базового типа. Однако при объявлении самого базового фиксированного типа диапазон значений следует задавать обязательно. Кроме того, диапазон значений подтипа не должен выходить за границы предыдущих диапазонов<sup>1)</sup>. Объявление подтипа SMALL\_FEE

subtype BAD\_PETTY\_CASH is SMALL\_FEE range 0.01 .. 50 000.00;

неправильно, так как верхняя граница диапазона значений этого подтипа выходит за диапазон значений для обозначения типа<sup>2)</sup>.

Уточнение диапазона индексов (index constraints), как отмечалось в разд. 2.2, применяется для указания дискретных границ значений индексов у регулярных типов. Диапазон индексов можно задавать только для тех типов или подтипов, которые еще не имеют его. Диапазоны индексов указываются при определении уточненных регулярных типов, для которых, следовательно, нельзя определять никакие подтипы.

**Пример.** Пусть имеется определение уточненного регулярного типа:

type RATES is array (1 .. J, K .. L) of FLOAT;

Тогда объявление

subtype BAD\_RATES is RATES(1 + 1 .. J - 1, K .. L);

будет неправильным, так как совокупность диапазонов индексов (1 .. J, K .. L) была задана уже для типа RATES, и, следовательно, указывать впоследствии новую совокупность диапазонов индексов (1 + 1 .. J - 1, K .. L) для подтипа BAD\_RATES уже было нельзя. Однако если есть объявление для неуточненного регулярного типа, например:

type MATRIX is array (INTEGER range <>, INTEGER range <>) of INTEGER;

то объявление подтипа

subtype POS\_MATRIX is MATRIX (1 + 1 .. J - 1, K .. L);

будет правильным.

Четвертый вид уточнений, которые можно употреблять в объявлениях подтипов, — это уточнение дискриминанта (discriminant constraint). Оно используется в

<sup>1)</sup> Здесь имеется в виду «каскадное» объявление подтипов, когда один подтип определяется на основе другого. — *Прим. перев.*

<sup>2)</sup> В данном случае — это подтип SMALL\_FEE. — *Прим. перев.*

объявлениях комбинированных типов. Как и в случае диапазона индексов, уточнение дискриминантов допустимо применять только для того типа или подтипа, у которого пока еще нет такого уточнения.

**Пример.** Рассмотрим тип PAGE\_FORMAT из предыдущего раздела. Он объявлен так:

```
type PAGE_FORMAT(NO_OF_LINES   : POSITIVE := 55;
                  NO_OF_COLUMNS : POSITIVE := 80)
    is
    record
        STD_HEADER : STRING(1 .. NO_OF_COLUMNS);
        STD_BODY   : PAGES(1 .. NO_OF_LINES,
                           1 .. NO_OF_COLUMNS);
        STD_LINES  : LINE_NUMBERS(1 .. NO_OF_LINES);
        STD_FOOTER : STRING(1 .. NO_OF_COLUMNS);
    end record;
```

Можно определить подтип для этого комбинированного типа, так как в самом типе отсутствует уточнение дискриминантов. Правильным будет следующее объявление:

```
subtype DRAFT_PAGES is PAGE_FORMAT (50, 80);
```

Здесь 50 и 80 – уточнение двух дискриминантов.

Другие примеры объявлений подтипов с использованием уточнений дискриминантов будут даны в гл. 4 после знакомства с вариантными комбинированными типами.

## 2.5.2. Производные типы

Подтипы принадлежат к исходному базовому типу и их значения – это значения, относящиеся к этому же исходному типу. В то же время при необходимости можно ввести так называемые *производные типы*, которые обладают новым набором значений, независимых от исходного набора, использованного при определении этих производных типов. Общая форма объявления производных типов такова:

```
type имя_производного_типа is new обозначение_типа;
```

Если есть уточнение, то

```
type имя_производного_типа is new обозначение_типа уточнение;
```

Уточнения, используемые для производных типов, имеют те же самые формы, что и для подтипов.

Обозначение\_типа в объявлении – это либо имя типа, либо имя подтипа. Базовый тип, указываемый в обозначении\_типа, называется в этом случае *родительским типом* для вводимого производного типа, а совокупность возможных значений производного типа – это копия совокупности возможных значений родительского типа. Операции, атрибуты, литералы и агрегаты (если они есть) родительского типа наследуются производным типом. Кроме того, допускаются явные преобразования величин родительского типа в величины производного типа и обратно.

**Примеры.** Рассмотрим некоторые примеры производных типов. В типе

```
type ACCOUNT is new INTEGER;
```

родительским типом является тип INTEGER. Значения производного типа ACCOUNT – это копии целых значений. Операции для объектов типа ACCOUNT дублируют операции для объектов типа INTEGER.

В объявлении

```
ACC1, ACC2: ACCOUNT;
```

объекты ACC1 и ACC2 относятся к типу ACCOUNT. В объявлении

```
NUM1, NUM2: INTEGER;
```

объекты NUM1 и NUM2 принадлежат к типу INTEGER. В операторе присваивания

```
ACC1 := ACC1 + 7;
```

при выполнении операции сложения для типа ACCOUNT принимают участие переменная ACC1 типа ACCOUNT и литерал 7, принадлежащий к универсальному целому типу. При использовании объектов универсального целого типа не требуется явного их преобразования к другим целым типам.

В операторе присваивания

```
NUM1 := ACC1 + NUM2;
```

употребляются производные типы. Этот оператор неправилен, так как нельзя складывать объекты двух разных типов.

В операторе присваивания

```
NUM1 := ACC1 + ACCOUNT(NUM2);
```

вначале осуществляется явное преобразование переменной NUM2 целого типа к типу ACCOUNT. Затем складываются две величины типа ACCOUNT. Из контекста можно определить, что эта операция относится к типу ACCOUNT. Потом значение типа ACCOUNT присваивается переменной типа INTEGER. Такое присваивание некорrekтно.

Вот еще примеры правильных операторов присваивания, в которых употребляются производные типы:

```
NUM1 := INTEGER(ACC1 + ACCOUNT(NUM2));
```

```
NUM1 := INTEGER(ACC1) + NUM2;
```

Более подробно правила преобразования типов будут изложены в гл. 6.

Теперь рассмотрим некоторые примеры производных типов, в которых используются уточнения. В объявлении

```
type ITEM is new INTEGER range 1 .. 9999;
```

ITEM — производный тип с уточнением диапазона значений. В объявлении

```
type FEE is delta 0.0001 range 0.00 .. 500_000.00;
```

FEE — это производный тип с указанием точности.

В объявлении

```
type PAYCHECK is new FEE delta 0.001;
```

```
I, J : INTEGER;
```

```
type IT_MATRIX is new MATRIX(I, J);
```

PAYCHECK — производный тип с указанием точности, а MATRIX — производный тип с уточнением диапазонов индексов. Тип MATRIX был определен ранее в настоящем разделе. В заключение рассмотрим объявление:

```
type PRINT_PAGE is new PAGE_FORMAT (59, 132);
```

Здесь PRINT\_PAGE — производный тип с уточнением дискриминантов.

Обратите внимание на то, что при использовании уточнения в определении производного типа вводится тип, значения которого являются копиями значений

родительского типа. В этом случае производный тип действует как подтип, значения которого отбираются среди значений родительского типа. Например, объявление

```
type XYZ is new FEE delta 0.01;
```

эквивалентно объявлению

```
type ABC is new FEE;
subtype XYZ is ABC delta 0.01;
```

Производные типы удобны для введения в программу логических и мнемонических отличительных особенностей путем определения совершенно разных типов для различных абстрактных понятий, что позволяет предотвратить их непреднамеренное перемешивание. Например, если определены типы

```
type BODY_TEMPERATURE is new FLOAT;
type BLOOD_COUNT is new FLOAT;
CURR_TEMP: BODY_TEMPERATURE;
CURR_BLOOD_COUNT: BLOOD_COUNT;
```

то нельзя перепутать переменные CURR\_TEMP и CURR\_BLOOD\_COUNT. Транслятор обнаружит этот вид ошибок, и программа не пойдет на выполнение. Преимущества использования производных типов и автоматически навязываемой ими дисциплины можно будет по достоинству оценить при разработке больших программ с сотнями объектов.

## УПРАЖНЕНИЯ

1. Напишите программу, которая вычисляет и выводит данные о годовых доходах, получаемых по ценным бумагам со скидкой, по задаваемой стоимости этих бумаг. Вычисления проводятся по формуле:

$$y = (rv - price)/price * b/dsm$$

Здесь приняты следующие обозначения:

y — годовой доход по ценным бумагам, хранящимся вплоть до срока выкупа (десятичное число, например, 9.85);

rv — сумма выкупа на 100 долл. номинальной стоимости ценной бумаги (обычно 100, но может быть, например, и 95);

price — стоимость ценной бумаги по курсу, деленная на 100 (например, 0.9805);

b — количество дней в году (365 или 366);

dsm — количество дней от даты оплаты до даты выкупа.

В каждой входной строке задается информация об одной ценной бумаге со скидкой. Формат — следующий:

Дата выкупа	Поз. 1–20
rv	Поз. 21–28 (два знака после десятичной точки)
price	Поз. 29–36 (четыре знака после десятичной точки)
b	Поз. 37–39 (целое число)
dsm	Поз. 40–42

Пример входной строки:

Позиция	12345678901234567890123456789	012345678901	2
Содержимое	December	27 1984	100.0 0.9875366055

Признаком конца входных данных служит строка со значением даты выкупа, равным «X»

». Для каждой ценной бумаги следует вывести дату ее выкупа и доход.

Требуется отдельно отобразить данные о ценной бумаге, приносящей наибольший доход, и о ценной бумаге, дающей наименьший доход. Эту информацию нужно разместить после всего списка бумаг. Корректность даты проверять не нужно.



2. Измените программу SHIP\_RATE из разд. 2.2.3 так, чтобы ACT\_DIST\_LIM и ACT\_WEIGHT\_LIM не были константами. Значения ACT\_DIST\_LIM и ACT\_WEIGHT\_LIM нужно вводить из первых двух входных строк данных. Необходимо проверять, указаны ли границы классов веса и расстояния в порядке возрастания их значений. Остальные входные строки не изменяются. Нужно ввести дополнительную проверку логичности вводимых данных о тарифах, т.е. значений CURR\_RATES: чем дальше отправляется груз, тем дороже должна быть стоимость его доставки.

3. Измените программу GRADES из разд. 2.2.3 таким образом, чтобы она выводила таблицу частот правильных ответов на вопросы.

4. Рассмотрим задачу составления платежной ведомости. Пусть федеральные налоги и налоги штатов для каждого из пятидесяти штатов исчисляются по различным шкалам. Величина налога находится в зависимости от доходов налогоплательщика. Диапазон доходов идентифицируется его нижней границей. Каждому диапазону доходов соответствует определенная ставка налога. Информация о ставках налогов вводится перед сведениями о служащих. Данные о ставках налогов имеют следующий формат: в первой строке располагается двухбуквенное обозначение штата (например, AL для штата Алабама), далее в двух позициях следует целое число, равное количеству диапазонов доходов, подлежащих налогообложению. В следующих нескольких строках, количество которых зависит от числа диапазонов доходов, размещаются данные о нижних границах диапазонов доходов и соответствующих им ставках налога. Каждое сведение о нижней границе и ставке налога занимает по 8 позиций строки. Таким образом, в одной строке максимально можно представить сведения о пяти диапазонах доходов. Эта информация задается для всех 50 штатов и для федерального правительства, идентифицируемого символами FG. Затем идут данные о каждом служащем в отдельности. Они имеют следующий формат:

Позиции	Данные
1-9	Номер по социальному страхованию
10-30	Фамилия служащего
31-38	Заработная плата за неделю (ее следует преобразовать в годовой доход, так как именно для него вычисляются налоги)
39-40	Количество иждивенцев

Последней строкой данных является строка со значением 999999999, размещенным в поле номера по социальному страхованию. Сумма дохода, подлежащего налогообложению, вычисляется как номинальная плата за вычетом 25 долл. на каждого иждивенца. Федеральный налог и налог штата исчисляются от этой суммы, а налог по социальному страхованию определяется как 7% от номинальной платы. Сумма, выдаваемая на руки, — это номинальная плата за вычетом всех налогов, т.е. налога по социальному страхованию, налога штата и федерального налога. Напишите программу, которая будет выводить фамилию каждого служащего и выдаваемую ему заработную плату.

5. Измените программу из упр. 1, считая, что дата оплаты и дата выкупа заданы в формате ГГММДД. Поле dsm заменено на два поля:

Дата оплаты	Поз. 40-45
Дата выкупа	Поз. 46-51

Дата из первых 20 позиций используется только для идентификации ценной бумаги. Необходимо проверять правильность двух введенных в этом задании дат. Они не могут приходиться на выходные дни. При проверке корректности дат вам может помочь текст программы DATE\_CONVERSION из разд. 2.4.3.

## Глава 3

# Ссылочные типы

### 3.1. ВВЕДЕНИЕ В ССЫЛОЧНЫЕ ТИПЫ

До сих пор переменные, объявляемые в декларативной части приводимых в данной книге подпрограмм, существовали и использовались в течение всего времени выполнения подпрограммы. В этом разделе вводится новый тип, называемый *ссылочным типом* (access type). Объекты этого типа называются *указателями*. Они создаются или уничтожаются в процессе выполнения подпрограммы.

Величины ссылочного типа обеспечивают доступ к другим объектам. Тип объектов, к которым можно обеспечить доступ при помощи данного ссылочного типа, задается при объявлении этого типа. Форма объявления ссылочного типа такова:

```
type имя_ссылочного_типа is access обозначение_типа;
```

Если имеются уточнения, то объявление принимает вид

```
type имя_ссылочного_типа is access обозначение_типа уточнение;
```

Обозначение\_типа — это имя типа или подтипа. В качестве уточнения можно использовать только уточнение границ диапазонов индексов или дискриминантов.

#### Пример

```
type NAME_STRINGS is new STRING (1 .. 20);  
type ACC_NAMES is access NAME_STRINGS;
```

Здесь ACC\_NAMES — имя ссылочного типа. Величины типа ACC\_NAMES указывают на объекты типа NAME\_STRINGS. Подразумевается, что тип NAME\_STRINGS объявлен раньше, чем выполнено объявление типа ACC\_NAMES.

#### 3.1.1. Особенности ссылочных типов

Во множестве значений каждого ссылочного типа обязательно встречается значение NULL. Указатель, равный NULL не указывает ни на какой другой объект иного типа. После объявления указателя ему присваивается значение NULL. Обратите внимание на то, что это — единственная ситуация, когда объекту языка Ада значение присваивается неявно.

#### Пример

```
AC_NAME_1, AC_NAME_2: ACC_NAMES;
```

Здесь записано объявление двух ссылочных переменных типа ACC\_NAMES. Их значения равны значению NULL. Это значение неявно присваивается переменным ACC\_NAME\_1 и ACC\_NAME\_2 во время трансляции программы.

Новые объекты ссылочного типа можно создавать во время выполнения программы, используя так называемые *генераторы*. Генератор обозначается словом new

(новый), за которым следует имя типа или подтипа (а при необходимости и уточнение). Значения новых объектов можно присваивать другим переменным того же типа<sup>1)</sup>.

### Пример

```
ACC_NAME_1 := new NAME_STRINGS;
```

Здесь создается объект типа NAME\_STRINGS. Значение ссылки на этот объект присвоено переменной ACC\_NAME\_1. Теперь значение этой переменной указывает на строку из 20 символов.

Основные операции для объектов ссылочного типа – это присваивание и проверка на равенство (=) или неравенство (/=). Следовательно, будет правильным оператор

```
ACC_NAME_2 := ACC_NAME_1;
```

Это означает, что указатель на строку из 20 символов, который размещен в ACC\_NAME\_1, будет размещен и в переменной ACC\_NAME\_2. Теперь обе переменные указывают на одну и ту же строку.

Проверка

```
ACC_NAME_2 = ACC_NAME_1
```

даст логическое значение TRUE только тогда, когда обе переменные указывают на один и тот же объект. В противном случае вырабатывается значение FALSE.

Для ссылки на те фактические объекты, которые указывает ссылочная переменная, используется особое обозначение. За именем ссылочного типа следуют точка и зарезервированное слово *all*<sup>2)</sup>.

Например, ACC\_NAME\_2.all – строковый объект типа NAME\_STRINGS. Тогда правильным будет оператор

```
ACC_NAME_2.all := "12345678901234567890";
```

Не существует никаких ограничений на типы объектов, доступ к которым может осуществляться посредством ссылочных типов. Они могут быть объектами перечисляемого, целого, регулярного, комбинированного, ссылочного или, как будет показано в дальнейшем, приватного типа.

**Пример.** Можно записать такие объявления:

```
type EMPLOYEE is
  record
    EMPL_NAME : STRING(1 .. 20);
    HRS_WORKED : INTEGER;
  end record;
type E_ACCESS is access EMPLOYEE;
I : E_ACCESS;
```

<sup>1)</sup> Генератор *new* создает анонимный объект некоторого типа X, доступ к которому осуществляется через указатель P типа X\_ACCESS, например:

```
type X is new STRING (1 .. 20);
type X_ACCESS is access X;
P : X_ACCESS;
P := new X;
```

В первой строке данного примера *new* не генератор, а объявление производного типа. В последней строке *new* – генератор. Он создает анонимный объект типа X, а ссылка на этот объект присваивается указателю P. Типом указателя P является X\_ACCESS, который обеспечивает возможность ссылки только на объекты типа X. Итак, генератор *new* создает и анонимный объект, и ссылку на него. – *Прим. ред.*

<sup>2)</sup> Это так называемая операция разыменования, когда вместо ссылки на объект получается его значение. – *Прим. перев.*

Значение I указывает на объекты типа EMPLOYEE. Запись I.all обозначает целостные структуры, а I.EMPL\_NAME относится только к компоненте объекта комбинированного типа, имеющей в данном примере строковый тип.

### 3.1.2. Программа с использованием ссылочных типов

Далее представлена программа, в которой используются переменные ссылочного типа. Эта программа является модификацией программы MAX3 из гл. 1. Она находит наибольшее среди трех целых чисел. На вход подаются сведения о трех служащих и количестве отработанных ими часов. Программа выводит фамилию наиболее загруженного служащего. Всего используются три входные строки (по одной на каждого служащего).

#### Программа ACCESS\_MAX3

```
with TEXT_IO; use TEXT_IO;
procedure ACCESS_MAX3 is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type EMPLOYEE is
    record
      EMPL_NAME   : STRING(1 .. 20);
      HRS_WORKED  : INTEGER;
    end record;
  type E_ACCESS is access EMPLOYEE;
  I,J,K,L : E_ACCESS;
begin
  I := new EMPLOYEE;
  -- Создан объект типа EMPLOYEE. Ссылочное
  -- значение, указывающее на него,
  -- присвоено переменной I. Заметьте, что
  -- I.EMPL_NAME не проинициализировано.
  J := new EMPLOYEE;
  K := new EMPLOYEE;
  GET(I.EMPL_NAME);GET(I.HRS_WORKED);
  SKIP_LINE;
  GET(J.EMPL_NAME);GET(J.HRS_WORKED);
  SKIP_LINE;
  GET(K.EMPL_NAME);GET(K.HRS_WORKED);
  NEW_LINE;
  if I.HRS_WORKED > J.HRS_WORKED
  then
    L := I;
    -- Сравнивается отработка у двух служащих.
    -- После этого L будет указывать на того
    -- служащего, кто отработал больше часов.
  else
    L := J;
  end if;
  if L.HRS_WORKED < K.HRS_WORKED
  then
    L := K;
  end if;
  PUT(" The hardest working is ")
  PUT(L.EMPL_NAME);
  PUT(" and he worked ");
  PUT(L.HRS_WORKED,5);
```

```

PUT(" hours ");
-- ПОМЕТКА для вставки оператора.
-- (Пояснения - в следующем разделе.)
and ACCESS_MAX3;

```

Рис. 3.1, а и 3.1, б демонстрируют значения указателя L для двух совокупностей входных данных.

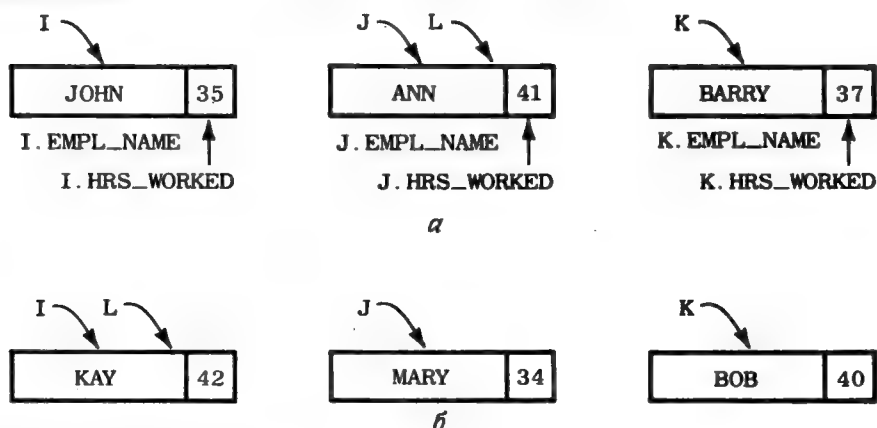


Рис. 3.1. Значения указателя L.

а — одна совокупность данных; б — другая совокупность данных.

### 3.1.3. Уничтожение объектов

Как показано в программе ACCESS\_MAX3, ссылочное значение, полученное с помощью генератора new, можно присваивать нескольким ссылочным переменным. Если говорить конкретно, то в программе ACCESS\_MAX3 на строку с наиболее загруженным служащим указывают две ссылочные переменные: L и одна из переменных I, J, K. До тех пор пока созданный объект доступен, т.е. на него указывает ссылочная переменная, память под объект распределена и он существует. Объект становится недоступным, когда никакая ссылочная переменная не может указать на него.

Что случится, если к памяти, занимаемой объектом, невозможно больше никакое обращение? В некоторых реализациях Ады эту память можно использовать в иных целях, т.е. она *освобождается*. В других реализациях этого не происходит.

Вернемся к программе ACCESS\_MAX3. Сделаем недоступными объекты, содержащие сведения о менее загруженных служащих. Их можно сделать недоступными, если в программу на отмеченное «место для вставки» поместить нижеследующий оператор if:

```

if L = I
then
  J := NULL; K := NULL;
-- Об'екты, на которые указывали переменные
-- J и K, становятся недоступными (к ним нельзя
-- больше обратиться). В зависимости от конкрет-
-- ной реализации Ады, память, которую занимали
-- эти об'екты, может или же не может быть
-- использована в дальнейшем.

```

```

elseif L = J
  then
    I := NULL; K := NULL;
  else
    I := NULL; J := NULL;
  end if;
-- Теперь имеется доступ (ссылка) только к одному
-- объекту; к нему можно обращаться с помощью L
-- и еще одной ссылочной переменной.

```

В связи с тем что память, занимаемая объектами, которые создаются при помощи генераторов `new`, изменяется во время выполнения программы, в Аде есть средства для явного уничтожения объектов. *Уничтожение*—это обратное действие по сравнению с созданием. Оно означает то, что ранее распределенная память становится доступной для иных целей. Явно уничтожить *ссылочный объект* (т.е. объект, доступ к которому осуществляется при помощи указателя) можно с помощью процедуры `FREE`, входящей в состав предопределенной (родовой) процедуры `UNCHECKED_DEALLOCATION`. Разумеется, при явном уничтожении ссылочных объектов следует проявлять большую осторожность, чтобы избежать случайного сохранения ссылочных переменных, указывающих на уже несуществующий объект.

### 3.1.4. Эффективное использование ссылочных типов

В некоторых ситуациях применение ссылочных типов позволяет достичь большей ясности программы и повысить ее эффективность. В качестве примера рассмотрим модифицированную программу `SHIP_RATE` (см. разд. 2.2.3). Назовем ее теперь `ACCESS_SHIP_RATE`. В ней сделаны следующие изменения. Вместо одной таблицы, как это было в `SHIP_RATE`, теперь из входного файла поступает несколько таблиц. В каждой таблице вводятся свои конкретные данные о категориях для расстояния доставки и веса груза. Таблица получает обозначение, состоящее из 10 символов. Если говорить более точно, то каждая таблица инициализируется с помощью четырех входных строк, первая из которых имеет следующий формат:

Позиции	Данные
1–10	Обозначение таблицы, например <code>OVERNIGHT</code> (ночная доставка)
11–34	Три числа (по восемь цифр в каждом), специфицирующие верхние границы для каждой категории расстояния. Числа следуют в порядке возрастания, например: <code>100.00 500.00 5000.00</code> Это означает, что для первой категории, соответствующей значению <code>CLOSE_BY</code> , расстояние находится в пределах от 0 до 100.0 миль; для третьей категории—от 500 до 5000 миль
35–66	Четыре числа (из восьми цифр каждое), расположенные в возрастающем порядке. Они задают верхние границы категорий веса груза

Каждая из трех остальных строк содержит по четыре действительных числа и представляет собой тариф на оплату доставки одного фунта груза в зависимости от его класса веса и класса расстояния. Формат их такой же, как для программы `SHIP_RATE`. Признаком конца таблиц служит строка первого рода, размещенная за последней таблицей, при этом в качестве обозначения таблицы используются символы `"XXXXXXXXXX"`. На вход поступает неизвестное заранее количество таблиц (однако это количество не может превышать 20). Остальные входные строки имеют почти тот же формат, что и для программы `SHIP_RATE`. В них добавлено только десяти-символьное поле с обозначением таблицы.

## Программа ACCESS\_SHIP\_RATE

```

with TEXT_IO; use TEXT_IO;
procedure ACCESS_SHIP_RATE is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type ALLOWED_WEIGHT is digits 10 range 0.00 ..
                                     7000.00;
  type ALLOWED_DISTANCE is digits 10 range 0.00 ..
                                     8000.00;
  package DISTANCE_IO is new
    FLOAT_IO(ALLOWED_DISTANCE);
  use DISTANCE_IO;
  package WEIGHT_IO is new FLOAT_IO(ALLOWED_WEIGHT);
  use WEIGHT_IO;
  type DISTANCE_CLASS is (CLOSE_BY, MED_DISTANCE,
                          LONG_DISTANCE);
  type CURR_DIST_CLASS : DISTANCE_CLASS;
  type WEIGHT_CLASS is
    ( LIGHT, MED_WEIGHT, MED_HEAVY, HEAVY);
  type CURR_WEIGHT_CLASS : WEIGHT_CLASS;
  type DIST_CATEGORIES is array ( DISTANCE_CLASS )
    of ALLOWED_DISTANCE;
  type WEIGHT_ARRAY is array ( WEIGHT_CLASS ) of
    ALLOWED_WEIGHT;
  type PRICES is delta 0.0001 range 0.000 ..
                                     5000.000;
  package PRICES_IO is new FIXED_IO(PRICES);
  use PRICES_IO;
  type SHIPPING_RATES is array ( DISTANCE_CLASS,
    WEIGHT_CLASS ) of PRICES;
  -- Следующая структура содержит всю информацию
  -- о таблице грузовых тарифов.
  type SHIP_TBL is
    record
      CATEGORY : STRING ( 1 .. 10 );
      ACT_DIST_LIM : DIST_CATEGORIES;
      ACT_WEIGHT_LIM : WEIGHT_ARRAY;
      CURR_RATES : SHIPPING_RATES;
    end record;
  type REGULAR_REC is
    record;
      ITEM_NO : INTEGER;
      ITEM_DISTANCE : ALLOWED_DISTANCE;
      ITEM_WEIGHT : ALLOWED_WEIGHT;
      ITEM_COST : PRICES;
      ITEM_SERVICE : STRING (1 .. 10);
    end record;
  INPUT_REC : REGULAR_REC;
  type SHIP_POINTER is access SHIP_TBL;
  type FULL_TBL is array (1 .. 20)
    of SHIP_POINTER;
  ACT_TBL : FULL_TBL;
  -- Можно создать до 20 указателей.
  TBL_CNT, CRT_CNT : INTEGER := 0;
begin

```

```

TBL_CNT := TBL_CNT + 1;
ACT_TBL (TBL_CNT) := new SHIP_TBL;
-- Создан объект типа SHIP_TBL, ссылочное
-- значение указывающее на него, помещается
-- в ACT_TBL(TBL_CNT).
-- Здесь добавлены дополнительные операторы
-- для обработки первой строки, в которой
-- указаны категория и верхние границы таблицы.
GET ( ACT_TBL( TBL_CNT ).CATEGORY );
-- Прочитать обозначение первой таблицы.
while ACT_TBL (TBL_CNT).CATEGORY /=
      "XXXXXXXXXX"
loop
  for JUNK_DIST in DISTANCE_CLASS
    loop
      GET(ACT_TBL(TBL_CNT).ACT_DIST_LIM(JUNK_DIST),
          8 );
      -- Введены верхние границы классов расстояния.
    end loop;
  for JUNK_WEIGHT in WEIGHT_CLASS
    loop
      GET(ACT_TBL(TBL_CNT).ACT_WEIGHT_LIM(
          JUNK_WEIGHT), 8);
      -- Введены верхние границы классов веса.
    end loop;
  SKIP_LINE;
  -- Здесь вставка кончается. Теперь следует
  -- текст, аналогичный тексту SHIP_RATE.
  for JUNK_DIST in DISTANCE_CLASS
    loop
      for JUNK_WEIGHT in WEIGHT_CLASS
        loop
          GET(ACT_TBL(TBL_CNT).CURR_RATES(
              JUNK_DIST,JUNK_WEIGHT), 7);
        end loop;
      SKIP_LINE;
    end loop;
  TBL_CNT := TBL_CNT + 1;
  ACT_TBL(TBL_CNT) := new SHIP_TBL;
  -- Создается другое ссылочное значение,
  -- указывающее на таблицу тарифов.
  GET(ACT_TBL(TBL_CNT).CATEGORY);
end loop;
GET(INPUT_REC.ITEM_NO,7);
while INPUT_REC.ITEM_NO /= 9999999
loop
  GET(INPUT_REC.ITEM_WEIGHT,7);
  GET(INPUT_REC.ITEM_DISTANCE,7);
  GET(INPUT_REC.ITEM_SERVICE);
  SKIP_LINE;
  -- Вставлен текст для поиска
  -- ITEM_SERVICE.
  CRT_CNT := 0;
  for I in 1 .. TBL_CNT
    loop
      if INPUT_REC.ITEM_SERVICE =
          ACT_TBL(TBL_CNT).CATEGORY

```



```

        then
            CRT_CNT := I;
        end if;
    end loop;
    -- Здесь текст вставки заканчивается. Операторы,
    -- расположенные ниже, выполняются только
    -- в том случае, когда значение ITEM_SERVICE
    -- входит в одну из таблиц.
    if CRT_CNT /= 0
    then
        for JUNK_DIST in reverse DISTANCE_CLASS
        loop
            if INPUT_REC.ITEM_DISTANCE >
                ACT_TBL(CRT_CNT).ACT_DIST_LIM(JUNK_DIST)
            then
                CURR_DIST_CLASS := JUNK_DIST;
            end if;
        end loop;
        for JUNK_WEIGHT in reverse WEIGHT_CLASS
        loop
            if INPUT_REC.ITEM_WEIGHT >
                ACT_TBL(CRT_CNT).ACT_WEIGHT_LIM(JUNK_WEIGHT)
            then
                CURR_WEIGHT_CLASS := JUNK_WEIGHT;
            end if;
        end loop;
        INPUT_REC.ITEM_COST :=
            PRICES(ACT_TBL(CRT_CNT).
                CURR_RATES(CURR_DIST_CLASS,
                CURR_WEIGHT_CLASS)
            * PRICES(INPUT_REC.ITEM_WEIGHT) );
        NEW_LINE;
        PUT(INPUT_REC.ITEM_COST,7,2);
    end if;
    SKIP_LINE;
    GET(ITEM_NO);
end loop;
end ACCESS_SHIP_RATE;

```

На рис. 3.2 показан возможный вид массива ссылочных переменных ACT\_TBL, созданного для трех типов таблиц и для произвольного числа видов отправляемых грузов.

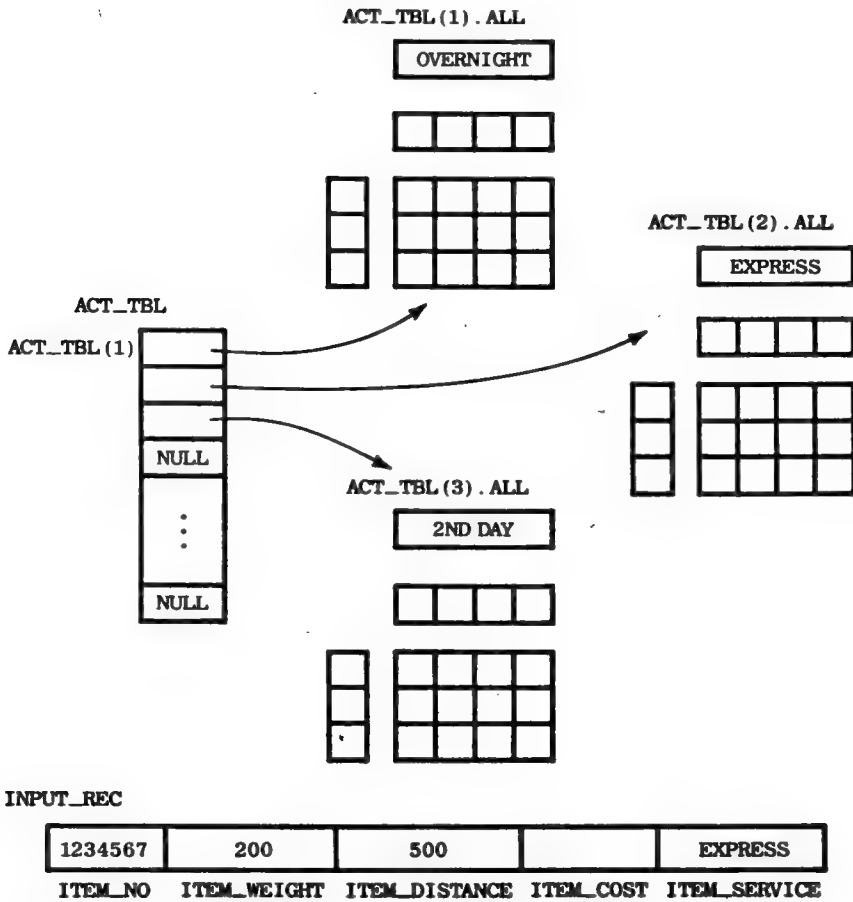
### 3.1.5. Уточнения

Как отмечалось ранее, при объявлении ссылочных типов допустимыми видами уточнений являются лишь уточнения дискриминантов и уточнения диапазонов индексов. Пусть, например, существует объявление

```
type NAME_POINTER is access.STRING (21..55);
```

Здесь в объявлении ссылочного типа присутствует уточнение диапазона индексов.

Если при объявлении задается доступ к неуточненному типу, то уточнение можно указать при создании объекта нужного типа с помощью генератора new.



**Рис. 3.2.** Значения массива ссылочных переменных ACT\_TBL для трех таблиц и тестовой строки входных данных.

**Пример.** Пусть имеются объявления

```
type LINE_POINTER is access STRING;
CURR_LINE: LINE_POINTER;
```

Тогда оператор

```
CURR_LINE := new STRING;
```

будет неправильным, так как STRING – не уточненный регулярный тип, а при создании объектов этого типа с помощью генератора new должно быть указано уточнение диапазона для индекса. Вместо этого следует, например, написать:

```
CURR_LINE := new STRING (8 .. 89);
```

где уточнение присутствует. А вот другие примеры:

```

type JOB_DESCRIPTION is array (NATURAL range <>)
  of STRING (1 .. 10);
type EMPLOYEE (JOB_NO : POSITIVE) is
  record
    L_NAME      : STRING (1 .. 30);
    JOBS_HELD   : JOB_DESCRIPTION (1 .. JOB_NO);
  end record;
type EMP_POINTER is access EMPLOYEE;

```

Здесь ссылочный тип был объявлен без уточнений, а в объявлении

```
type EMP_CONSTR_POINTER is access EMPLOYEE (20);
```

имеется уточнение дискриминанта. В заключение рассмотрим объявления:

```

JOHN_PAUL : EMP_POINTER;
MARY_LOU : EMP_CONSTR_POINTER;

```

Если есть такие объявления, то каждый раз при создании объекта типа EMP\_POINTER следует указывать уточнение дискриминанта, например:

```
JOHN_PAUL := new EMPLOYEE (5);
```

Однако при создании объектов типа EMP\_CONSTR\_POINTER уточнение дискриминанта не задается:

```
MARY_LOU := new EMPLOYEE;
```

Если уточнение указано при объявлении типа, то при создании объектов этого типа нельзя задавать уточнение еще раз.

Программу ACCESS\_SHIP\_RATE можно было бы написать с использованием динамических массивов, а не ссылочных типов. Применение ссылочных типов будет оправданным, если в результате этого программа окажется более ясной, будет выполнять более общие функции, станет более надежной или позволит более эффективно использовать память. Эти преимущества станут еще более очевидными, когда решаемую задачу можно будет естественно представить в рекурсивной форме, что будет показано в следующем разделе.

## 3.2. РЕКУРСИВНЫЕ ОБЪЯВЛЕНИЯ ССЫЛОЧНЫХ ТИПОВ

Не существует ограничения для типов объектов, на которые указывает переменная ссылочного типа. Поэтому та компонента сложного объекта, на которую указывает некоторая переменная ссылочного типа, сама может являться переменной того же ссылочного типа. В этом случае требуется записать так называемое незавершенное объявление для нужного типа, к которому производится ссылка, за ним объявление соответствующего ссылочного типа и далее полное объявление для требуемого типа.

### 3.2.1. Незавершенные объявления типа

*Незавершенное объявление типа* в языке Ада имеет форму:

```
type идентификатор возможная_дискриминантная_часть;
```

**Пример.** Вот пример незавершенного объявления типа:

```
type PAYROLL_REC;
```

Далее могут следовать прочие объявления:

```
type PAY_ACCESS is access PAYROLL_REC;
```

Здесь переменные ссылочного типа PAY\_ACCESS предназначены для доступа к объектам типа PAYROLL\_REC. Далее опять могут следовать не относящиеся к примеру объявления. Потом помещается объявление:

```
type PAYROLL_REC is
  record
    EMP_NAME : STRING (1 .. 20);
    EMP_ID   : STRING (1 .. 9);
    EMP_PAY  : FLOAT;
    EMP_NEXT : PAY_ACCESS;
  end record;
```

Здесь компонента EMP\_NEXT комбинированного типа PAYROLL\_REC, к которому производится доступ с помощью ссылочной переменной типа PAY\_ACCESS, является ссылочной переменной того же типа PAY\_ACCESS. Далее располагаются объявления переменных:

```
CURR_PTR, PREV_PTR: PAY_ACCESS;
HOLD_REC: PAYROLL_REC;
```

Покажем примеры корректных операторов, которые создают объекты типа PAYROLL\_REC:

```
CURR_PTR := new PAYROLL_REC;
```

Эта строка создает объект типа PAYROLL\_REC, на который указывает переменная CURR\_PTR типа PAY\_ACCESS.

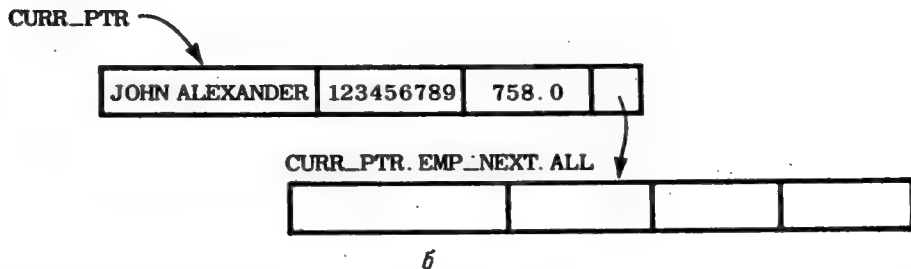
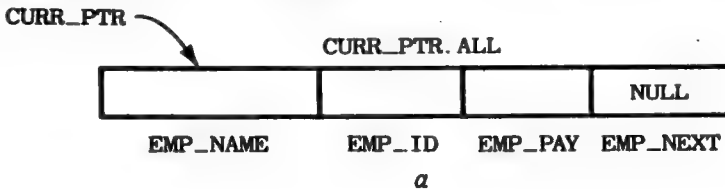


Рис. 3.3. Объекты типа PAY\_ACCESS.

a—после первого выполнения оператора CURR\_PTR := new PAYROLL\_REC; б—после первого выполнения оператора CURR\_PTR.EMP\_NEXT := new PAYROLL\_REC.

На рис. 3.3,а показаны значения различных компонент после выполнения этого оператора. Вот еще примеры:

```
CURR_PTR.EMP_NAME := "JOHN ALEXANDER";
CURR_PTR.EMP_ID   := "123456789";
CURR_PTR.EMP_PAY  := 758.00;
CURR_PTR.EMP_NEXT := new PAYROLL_REC;
```

На рис. 3.3,6 показаны значения различных компонент, на которые ссылается указатель CURR\_PTR, после выполнения приведенных выше операторов.

Оператор

```
PREV_PTR := CURR_PTR;
```

выполняет подготовку к присваиванию переменной CURR\_PTR нового значения. Присваивание выполняется операторами

```
CURR_PTR := CURR_PTR.EMP_NEXT;  
CURR_PTR.EMP_NAME := "EUGENE SORENSEN"  
CURR_PTR.EMP_ID := "222222222";  
CURR_PTR.EMP_PAY := 800.00;  
CURR_PTR.EMP_NEXT := new PAYROLL_REC;  
HOLD_REC := ("MARY MURPHY", "555555555",  
            850.00, NULL);
```

Обратите внимание на то, что переменная HOLD\_REC типа PAYROLL\_REC инициализируется при помощи позиционного агрегата комбинированного типа. Присваивание завершается оператором

```
CURR_PTR.EMP_NEXT.all := HOLD_REC;
```

Структура, к которой обращается переменная ссылочного типа CURR\_PTR.EMP\_NEXT.all, проинициализирована содержимым переменной HOLD\_REC. Однако было бы неверным написать:

```
CURR_PTR.EMP_NEXT := HOLD_REC;
```

поскольку значение переменной комбинированного типа нельзя присваивать переменной ссылочного типа.

Заметьте также, что к структуре, созданной первой, можно получить доступ через PREV\_PTR, ко второй структуре — через PREV\_PTR.EMP\_NEXT и через CURR\_PTR, а к третьей структуре — через CURR\_PTR.EMP\_NEXT. Итоговые значения переменных отображены на рис. 3.4.

### 3.2.2. Взаимная зависимость

Можно сказать, что объявления, подобные объявлению для типа PAY\_ACCESS, называются *рекурсивными объявлениями типов*, так как в своем определении они ссылаются сами на себя. Можно ввести ссылочные типы, которые ссылаются друг на друга, установив тем самым *взаимную зависимость*.

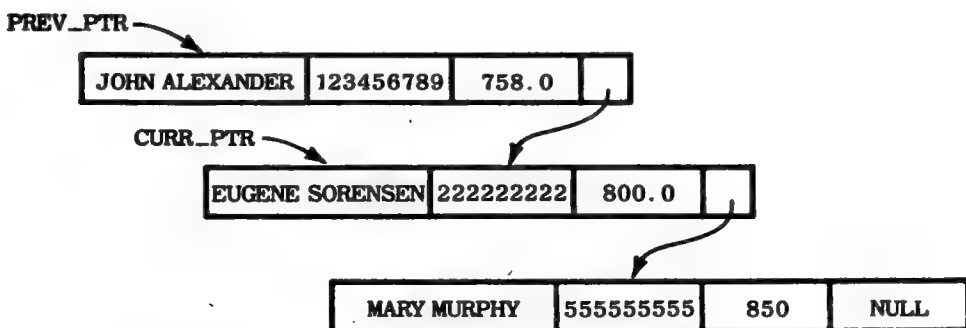
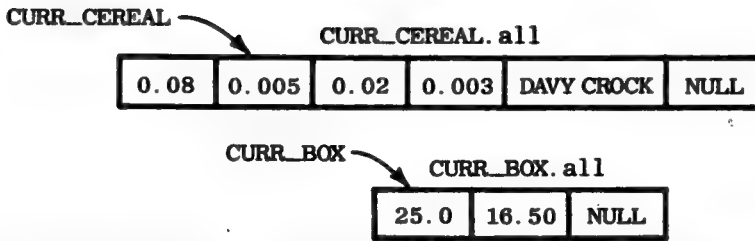


Рис. 3.4. Окончательные значения объектов типа PAY\_ACCESS.



**Рис. 3.5.** Значения переменных `CURR_CEREAL` и `CURR_BOX` до установления взаимной зависимости.

**Пример.** Запишем два незавершенных объявления типа:

```
type CEREAL;
type PACKAGE_BOX;
```

Далее можно записать

```
type CEREAL_PTR is access CEREAL;
type PACK_BOX_PTR is access PACKAGE_BOX;
```

Здесь объявляются два ссылочных типа, при помощи которых можно обращаться к типам, незавершенные объявления которых были даны вначале. Далее запишем:

```
type CEREAL is
  record
    SUGAR_CONT      : FLOAT;
    SALT_CONT       : FLOAT;
    FIBER_CONT      : FLOAT;
    VITAMIN_C_CONT  : FLOAT;
    ADV_TYPE        : STRING ( 1 .. 10 );
    PACKING_INSTR   : PACK_BOX_PTR;
  end record;
type PACKAGE_BOX is
  record
    SHIP_VOLUME     : FLOAT;
    SHIP_WEIGHT     : FLOAT;
    SHIP_CONTENT    : CEREAL_PTR;
  end record;
```

Взаимная зависимость возникает здесь следующим образом. Тип `CEREAL` ссылается на тип `PACKAGE_BOX` через ссылочную переменную `PACKING_INSTR`, а тип `PACKAGE_BOX` ссылается на тип `CEREAL` через ссылочную переменную `SHIP_CONTENT`. Далее:

```
CURR_CEREAL : CEREAL_PTR;
CURR_BOX : PACK_BOX_PTR;
```

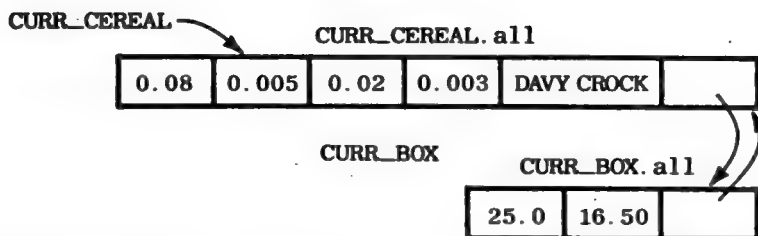
А вот примеры операторов, в которых используется эта взаимная зависимость:

```
CURR_CEREAL := new CEREAL;
CURR_CEREAL.all := (0.08, 0.005, 0.02, 0.003, "DAVY CROCK", NULL);
```

Здесь проинициализирована структура `CURR_CEREAL.all`. Продолжим:

```
CURR_BOX := new PACKAGE_BOX;
CURR_BOX.all := (25.00, 16.50, NULL);
```

Пока две ссылочные переменные указывают на объекты, которые не ссылаются друг на друга (рис. 3.5). С помощью операторов:



**Рис. 3.6.** Значения переменных `CURR_CEREAL` и `CURR_BOX` после установления взаимной зависимости.

```
CURR_CEREAL.PACKING_INSTR := CURR_BOX;
CURR_BOX.SHIP_CONTENT := CURR_CEREAL;
```

устанавливается взаимная зависимость (рис. 3.6).

### 3.2.3. Программа, использующая рекурсивное объявление типа

Следующая программа иллюстрирует понятия рекурсивных определений типов и взаимной зависимости. Эта программа является модификацией программы `GRADES` из разд. 2.2.3. Пусть имеется несколько видов контрольных работ, каждый из которых обозначается пятисимвольной строкой, например `MATH1` для математики № 1. Информация, необходимая для оценки каждой контрольной работы, размещается в двух входных строках. Первая строка имеет формат:

Позиции	Данные
1–5	Наименование предмета ( <code>MATH1</code> , <code>ENGL5</code> и т. д.)
6–7	Число вопросов в контрольной работе (от 20 до 50)

Вторая строка содержит последовательность цифр от 1 до 5. Количество этих цифр равно числу вопросов, указанному в первой строке. Цифры представляют собой номера правильных ответов. Это «ключи ответов». Признаком конца информации о ключах ответов служит строка, содержащая символы «XXXXX» в поле наименования предмета.

Остальные строки входного файла содержат сведения об ответах студентов. В десяти начальных позициях каждой строки помещен личный номер студента, в пяти следующих позициях записано наименование предмета. Сами ответы (точнее номера выбранных студентом вариантов ответов) начинаются с позиции 16. Как и в первоначальной версии программы `GRADES`, признаком конца последовательности строк с ответами служит запись, содержащая 9999999999 в качестве личного номера студента. Выходные данные программы такие же, как и раньше. Для каждой считанной строки с ответами студента отображается его фамилия и количество правильных ответов.

#### Программа `ACCESS_GRADES`

```
with TEXT_IO; use TEXT_IO;
procedure ACCESS_GRADES is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
```

```

type CHOICES is range 1 .. 5;
type POSSIBLE_QUESTIONS is range 1 .. 50;
package CHO_IO is new INTEGER_IO(CHOICES);
use CHO_IO;
package POSS_IO is new
    INTEGER_IO(POSSIBLE_QUESTIONS);
use POSS_IO;
DNO_QUESTIONS : POSSIBLE_QUESTIONS := 50;
type ANSWERS is array ( 1 .. DNO_QUESTIONS ) of
    CHOICES;

type TEST_KEY;
-- Незавершенное об'явление типа. Оно необходимо
-- здесь для рекурсивного об'явления типа.
type TEST_KEY_PTR is access TEST_KEY;
type TEST_KEY is
    record
        SUBJ      : STRING(1 .. 5);
        NO_QUESTIONS : POSSIBLE_QUESTIONS;
        KEY_ANSWERS : ANSWERS;
        NEXT_TEST  : TEST_KEY_PTR;
    end record;
CURR_TEST, START_TEST : TEST_KEY_PTR;
GOOD_ANSWERS : INTEGER ;
type IN_REC is
    record;
        STUDENT_ID : STRING ( 1 .. 10);
        SUBJECT_ID : STRING ( 1 .. 5 );
        STUDENT_ANSWERS : ANSWERS;
    end record;
CURR_REC : IN_REC;
begin
    CURR_TEST := new TEST_KEY;
    GET(CURR_TEST.SUBJ);
    while CURR_TEST.SUBJ /= "XXXXX"
    loop
        -- Здесь строится список контрольных работ
        -- Это односвязный список. На начало списка
        -- указывает переменная START_TEST.
        GET (CURR_TEST.NO_QUESTIONS,2);
        DNO_QUESTIONS := CURR_TEST.NO_QUESTIONS;
        SKIP_LINE;
        for I in 1 .. CURR_TEST.NO_QUESTIONS
            loop
                GET(CURR_TEST.KEY_ANSWERS (I), 1);
            end loop;
        if START_TEST = NULL
            then
                -- START_TEST равно NULL только после чтения
                -- первой строки данных. Первое прочитанное
                -- название контрольной работы заносится в
                -- список.
                START_TEST := CURR_TEST;
            else
                CURR_TEST.NEXT_TEST := START_TEST;
                START_TEST := CURR_TEST
                -- Каждая новая контрольная работа ставится
                -- в начало списка, она занимает это место.

```



```

-- до начала обработки следующей контроль-
-- ной работы.
end if;
SKIP_LINE;
CURR_TEST := new TEST_KEY;
GET(CURR_TEST.SUBJ);
end loop;
SKIP_LINE;
GET ( CURR_REC.STUDENT_ID );
while CURR_REC.STUDENT_ID /= "9999999999"
loop
GET (CURR_REC.SUBJECT_ID);
CURR_TEST := START_TEST;
while CURR_TEST /= NULL or
CURR_TEST.SUBJ /= CURR_REC.SUBJECT_ID
loop
-- Поиск в списке предмета, по которому
-- проводится контрольная работа.
CURR_TEST := CURR_TEST.NEXT_TEST;
end loop;
if CURR_TEST.SUBJ = CURR_REC.SUBJECT_ID
then
-- Если наименование контрольной работы
-- правильное, то будет найдено название
-- предмета. В противном случае такого
-- названия предмета нет в списке.
GOOD_ANSWERS := 0;
for J in 1 .. CURR_TEST.NO_QUESTIONS
loop
GET(CURR_REC.STUDENT_ANSWERS(J), 1);
if CURR_REC.STUDENT_ANSWERS(J) =
CURR_TEST.KEY_ANSWERS (J)
then
GOOD_ANSWERS := GOOD_ANSWERS + 1 ;
end if;
end loop;
NEW_LINE;
PUT ( " The number of good answers is " );
PUT ( GOOD_ANSWERS, 5);
PUT ( " for the id ");
PUT ( CURR_REC.STUDENT_ID );
NEW_LINE;
else
PUT(" No such subject: ");
PUT(CURR_REC.SUBJECT_ID);
end if;
SKIP_LINE;
GET ( CURR_REC.STUDENT_ID ) ;
end loop;
end ACCESS_GRADES;

```

На рис. 3.7 показан список ключей ответов для трех предметов (MATH1, ENGL1 и COMP1) и дан пример ответов студента в виде записи CURR\_REC.

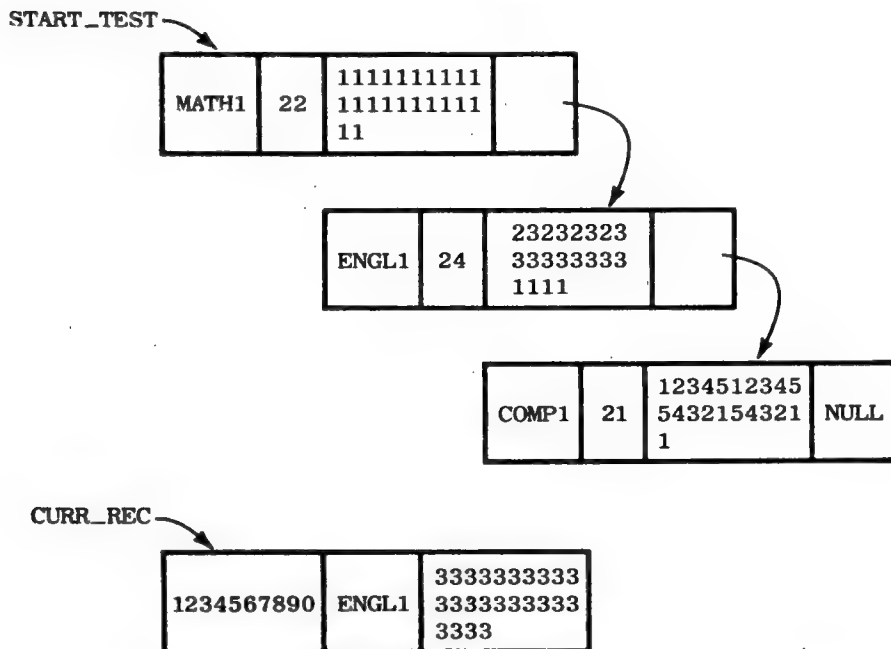


Рис. 3.7. Тестовые данные: список контрольных работ и строка с ответами студента.

### 3.2.4. Программа, в которой используется взаимная зависимость

В следующей программе дается пример взаимной зависимости, реализованной с помощью ссылочных типов. Программа выводит список оценочных баллов для лошадей, участвующих в предстоящем заезде на скачках.

Входная информация задается в парах строк. В первой из них приводятся данные о лошади, а во второй — о жокее.

Информация о лошади имеет такой формат:

Позиции	Данные
1-10	Кличка лошади
11-13	Наилучший результат (в с)
14-15	Число побед в последних десяти заездах

Данные о жокее вводятся в следующем формате:

Позиции	Данные
1-10	Фамилия жокея
11-13	Вес жокея (в фунтах)
14-15	Число побед в последних десяти заездах

Балл, который получает лошадь, будем вычислять в соответствии с выражением

$$\text{балл} = (\text{наилучший\_результат\_лошади}) - 5 * (\text{число\_побед\_лошади}) -$$

5\* (число\_побед\_жокея) +  
2\* (вес\_жокея)

(без какого-либо «научного» обоснования примененной формулы).

В заездах участвует заранее неизвестное число лошадей. Признаком конца данных служит запись с именем лошади CALIGULA. Список лошадей, наездников и баллов следует выдать в возрастающем (для баллов) порядке.

### Программа RACES

```
with TEXT_IO; use TEXT_IO;
procedure RACES is
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
type HORSE;
type JOCKEY;
-- Незавершенные определения типов требуются для
-- введения двух взаимно зависимых типов.
type HORSE_PTR is access HORSE;
type JOCKEY_PTR is access JOCKEY;
type HORSE is
  record
    H_NAME : STRING ( 1 .. 10 );
    WIN_TIME : INTEGER;
    WIN_RACE : INTEGER;
    RIDER : JOCKEY_PTR;
    -- Переменная RIDER указывает на об'екты типа
    -- JOCKEY.
    PAIR_RANK : INTEGER;
    NXT_HORSE : HORSE_PTR;
  end record;
type JOCKEY is
  record
    J_NAME : STRING ( 1 .. 10 );
    J_WEIGHT : INTEGER;
    J_WINS : INTEGER;
    -- Переменная J_WINS указывает на
    -- об'екты типа HORSE.
    H_TO_RIDE : HORSE_PTR;
  end record;
CURR_HORSE_PTR, TOP_HORSE_PTR, ANY_HORSE_PTR,
PREV_HORSE_PTR : HORSE_PTR;
CURR_JOCKEY_PTR : JOCKEY_PTR;

begin
  CURR_HORSE_PTR := new HORSE;
  GET(CURR_HORSE_PTR.H_NAME);
  while CURR_HORSE_PTR.H_NAME /= "CALIGULA"
  loop
    GET(CURR_HORSE_PTR.WIN_TIME,3);
    GET(CURR_HORSE_PTR.WIN_RACE,2);
    SKIP_LINE;
    CURR_JOCKEY_PTR := new JOCKEY;
    GET(CURR_JOCKEY_PTR.J_NAME);
    GET(CURR_JOCKEY_PTR.J_WEIGHT,3);
    GET(CURR_JOCKEY_PTR.J_WINS,2);
    SKIP_LINE;
```

```

CURR_HORSE_PTR.RIDER := CURR_JOCKEY_PTR;
CURR_JOCKEY_PTR.H_TO_RIDE := CURR_HORSE_PTR;
-- Эти присваивания устанавливают взаимную
-- зависимость.
CURR_HORSE_PTR.PAIR_RANK :=
  CURR_HORSE_PTR.WIN_TIME -
  5 * (CURR_HORSE_PTR.WIN_RACE +
        CURR_JOCKEY_PTR.J_WINS ) +
  2 * CURR_JOCKEY_PTR.J_WEIGHT;
-- После вычисления оценки в баллах, пара
-- лошадь-жокей помещается в список, форми-
-- руемый в порядке возрастания баллов.
ANY_HORSE_PTR := TOP_HORSE_PTR;
while ANY_HORSE_PTR.PAIR_RANK <
  CURR_HORSE_PTR.PAIR_RANK
  or
  ANY_HORSE_PTR /= NULL
  loop
    PREV_HORSE_PTR := ANY_HORSE_PTR;
    ANY_HORSE_PTR := ANY_HORSE_PTR.NXT_HORSE;
  end loop;
if ANY_HORSE_PTR = TOP_HORSE_PTR
  then
    CURR_HORSE_PTR.NXT_HORSE := TOP_HORSE_PTR;
    TOP_HORSE_PTR := CURR_HORSE_PTR;
  else
    PREV_HORSE_PTR.NXT_HORSE := CURR_HORSE_PTR;
    CURR_HORSE_PTR.NXT_HORSE := ANY_HORSE_PTR;
  end if;
GET(CURR_HORSE_PTR.H_NAME);
end loop;
ANY_HORSE_PTR := TOP_HORSE_PTR;
-- Теперь отобразим список в порядке возрастания
-- оценочных баллов.
while ANY_HORSE_PTR /= NULL
  loop
    PUT(ANY_HORSE_PTR.H_NAME);
    CURR_JOCKEY_PTR := ANY_HORSE_PTR.RIDER;
    PUT(CURR_JOCKEY_PTR.J_NAME);
    PUT(ANY_HORSE_PTR.PAIR_RANK);
    ANY_HORSE_PTR := ANY_HORSE_PTR.NXT_HORSE;
  end loop;
end RACES;

```

Пример возможных итоговых значений для трех лошадей и трех жокеев показан на рис. 3.8.

Полностью оценить преимущества, предоставляемые рекурсивными объявлениями типов и взаимной зависимостью, можно только после изучения рекурсивных вызовов подпрограмм, рассмотренных в гл. 5.

## УПРАЖНЕНИЯ

1. Измените программу ACCESS\_CHIP\_RATE так, чтобы сохранялись итоговые данные о видах отгруженных товаров, классифицированные по способу доставки (т.е. по наименованию таблицы). После обработки всех позиций товаров следует вывести эти итоговые данные и соответствующие названия таблиц.

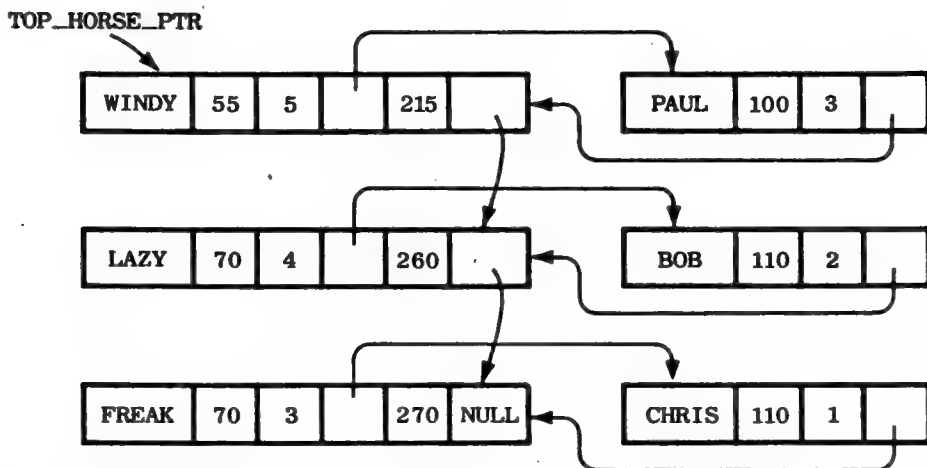


Рис. 3.8. Итоговые значения структур для примера скачек с тремя лошадьми и тремя жокеями.

2. Модифицируйте программу ACCESS\_GRADES так, чтобы для каждого предмета в возрастающем порядке строился список студентов и их оценок. Используйте ссылочные типы. После ввода данных о всех студентах следует вывести для каждого предмета список студентов в порядке возрастания полученных ими оценок.

3. Измените программу RACES таким образом, чтобы для объектов типа JOCKEY была добавлена дополнительная компонента, указывающая на объект того же самого типа. Эту компоненту нужно использовать для сортировки объектов типа JOCKEY по параметру J\_WEIGHT, характеризующему вес жокея. На выходе должны быть получены два вида списков наездников и их лошадей. Порядок следования данных в этих списках вначале должен соответствовать вычисленным баллам, а потом — весам жокеев.

# Прочие операторы языка Ада и комбинированные типы с вариантами

## 4.1. ПРОСТЫЕ И СОСТАВНЫЕ ОПЕРАТОРЫ АДЫ

Как отмечалось в гл. 1, в языке Ада есть два вида операторов: простые и составные. Если оператор не содержит в себе иных операторов, то это — *простой оператор*. *Составные операторы*, напротив, содержат в своем составе другие операторы.

До сих пор мы пользовались только двумя простыми операторами: операторами присваивания и операторами вызова процедур. Составные операторы, которые мы употребляли в первых трех главах — это операторы *if* (если) и две разновидности оператора цикла (*loop*): оператор *for* (для) и оператор *while* (пока).

В данном разделе будут рассмотрены другие простые операторы: пустой оператор (*null*), оператор выхода (*exit*) и оператор перехода (*goto*). Еще об одном простом операторе, операторе возврата (*return*), будет рассказано в следующей главе — там, где будут даны сведения о подпрограммах. В число остальных простых операторов языка Ада входят оператор прекращения задачи (*abort*), оператор задержки (*delay*), оператор обращения ко входу (*entry call*). (С ними читатель познакомится в гл. 10, посвященной параллельным процессам.) В гл. 11 излагается материал об исключительных ситуациях, там будет описан оператор возбуждения исключительных ситуаций (*raise*). Кроме того, существует специальный оператор включения кода (*code statement*), связанный с реализацией операторов языка низкого уровня (возможно, уровня языка ассемблера).

В данном разделе читатель познакомится с двумя новыми составными операторами: с оператором выбора (*case*) и с оператором блока (*block statement*). Будет также представлена третья (и последняя) разновидность оператора цикла. В языке Ада есть еще два составных оператора: оператор приема (*accept*) и оператор отбора (*select*). Они обсуждаются в гл. 10.

### 4.1.1. Оператор выбора *case*

*Оператор case* обеспечивает выбор и исполнение только одной из нескольких возможных альтернатив в зависимости от значения выражения, принадлежащего к дискретному типу. Форма оператора выбора такова:

```
case селектирующее_выражение is  
  when условие_выбора = > последовательность_операторов  
  ---Здесь могут располагаться другие альтернативы.  
  when условие_выбора = > последовательность_операторов  
end case;
```

Селектирующее\_выражение, стоящее после зарезервированного слова *case*, должно быть дискретного типа, т. е. значение, получаемое в результате вычисления этого выражения, должно принадлежать к целому или перечисляемому типу. В зависимости от значения селектирующего\_выражения будет выполняться одна из последовательностей операторов, располагающихся после условий\_выбора. Конкретно, будет выполнена та последовательность операторов, условие\_выбора для которой согласу-

ется с текущим значением селектирующего выражения, получаемым при выполнении программы в данный момент времени.

Условие выбора, стоящее после зарезервированного слова `when`, должно представлять собой статическое выражение дискретного типа или статическое выражение с дискретным диапазоном. Можно также в качестве последней по порядку альтернативы использовать условие выбора, обозначаемое зарезервированным словом `others` (другие). Эта альтернатива охватывает все остальные возможные значения селектирующего выражения, не заданные в предыдущих условиях выбора. Кроме того, в каждом условии выбора разрешается задавать несколько условий, разделенных символом `|`. В этом случае, если хотя бы одно из указанных в альтернативе условий соответствует значению селектирующего выражения, то будет выполняться выбираемая этим условием последовательность операторов.

**Пример.** Пусть переменная `SOME_CHAR` относится к типу `CHARACTER`. Тогда можно написать такой оператор выбора:

```
case SOME_CHAR is
  when 'A' .. 'Z' => PUT(" Uppercase letter ");
  when 'a' .. 'z' => PUT(" Lowercase letter ");
  when '0' .. '9' => PUT(" Digit ");
  when '+' | '-' => PUT(" Signs ");
  when others    => PUT(" Other characters ");
end case;
```

После зарезервированного слова `case` стоит простейшее выражение `SOME_CHAR`, принадлежащее к перечисляемому типу `CHARACTER`. Ясно, что `CHARACTER` — это дискретный тип. Перечислим представленные в этом операторе условия выбора.

Условие	Описание
'A' .. 'Z'	Статический диапазон
'a' .. 'z'	Статический диапазон
'0' .. '9'	Статический диапазон
'+'	Символьный литерал (элементарное выражение)
'-'	Символьный литерал (элементарное выражение)
Others (Прочие)	Охватывает все остальные значения, которые может принимать переменная <code>SOME_CHAR</code> , не входящие в предыдущие условия выбора

Заметьте, что

```
when '+' | '-' => PUT("Signs");
```

означает, что если `SOME_CHAR` равно `'+'` или `'-'`, то будет выведено сообщение "Signs" («Знаки»).

Для любой пары альтернатив, входящих в один и тот же оператор выбора `case`, нельзя записывать в условиях выбора альтернатив одинаковые значения. Например, оператор

```
case SOME_CHAR is
  when 'A' .. 'M' => PUT(" First half ");
  when 'L' .. 'Z' => PUT(" Second half ");
end case;
```

ошибочен, так как статические диапазоны `'A' .. 'M'` и `'L' .. 'Z'` пересекаются, в них входят общие значения `'L'` и `'M'`.

Разумеется, вместо оператора выбора `case` всегда можно употребить условный оператор `if`. Так, первый оператор `case` данного примера эквивалентен такому услов-

ному оператору:

```

if SOME_CHAR in 'A' .. 'Z'
then
  PUT(" Uppercase letter ");
elsif SOME_CHAR in 'a' .. 'z'
then
  PUT(" Lowercase letter ");
elsif SOME_CHAR in '0' .. '9'
then
  PUT(" Digit ");
elsif SOME_CHAR = '+' or SOME_CHAR = '-'
then
  PUT(" Signs ");
else
  PUT(" Other characters ");
end if;
```

Зачастую, как свидетельствует этот пример, оператор выбора case более нагляден, чем условный оператор if. Это особенно справедливо при большом количестве дискретных альтернатив.

Обратите внимание на то, что совокупность значений выражений в условиях выбора должна покрывать все возможные значения селектирующего выражения, стоящего после зарезервированного слова case. Поэтому, например, было бы ошибкой удалить из предыдущего примера оператор выбора альтернатив others.

### 4.1.2. Метки

Метки в программах на языке Ада можно располагать перед любым простым и составным оператором. Метка — это идентификатор, заключенный в двойные угловые скобки. Например, можно написать:

```

«JUST_IN_CASE» if I < 0
  then I := -I; end if;
```

Здесь метка «JUST\_IN\_CASE» идентифицирует оператор if. Перед оператором можно поставить и несколько меток. Будет правильной такая строка:

```

«GOOD» «BON» «GUT» K := K + 1;
```

Метки в языке Ада явно не объявляются. Считается, что они объявлены неявно в конце декларативной части самого внутреннего тела подпрограммы, содержащего эти метки. Другим неявно объявляемым объектом, с которым мы уже имели дело, является параметр\_цикла в операторе for. Однако параметр\_цикла неявно объявлялся как локальная переменная оператора цикла. Поэтому параметр\_цикла известен в более узкой области программы, чем метка. Здесь есть два аспекта. Во-первых, метка известна во всем (самом внутреннем) охватывающем ее теле подпрограммы, а параметр\_цикла известен только внутри оператора цикла. Во-вторых, неявное объявление метки вступает в силу перед началом выполнения каких-либо операторов подпрограммы, а параметр цикла существует только во время выполнения оператора цикла. Как будет показано в гл. 7 и 10, метки можно также применять в телах пакетов и задач.



### 4.1.3. Оператор перехода goto

Передача управления на помеченный оператор выполняется с помощью *оператора перехода goto*. Общая форма оператора перехода такова:

```
goto имя_метки;
```

Например:

```
goto GOOD;
```

—правильный оператор перехода.

Сфера применения оператора перехода в языке Ада довольно ограничена. Этот оператор не должен передавать управление из программы или вовнутрь ее. Не разрешается также передача управления внутрь составного оператора, к примеру внутрь операторов if, case или loop. Запрещается передача управления по оператору goto от других программных сегментов — пакетов или задач. Внутри оператора if или case оператор перехода goto не может передавать управление от одной альтернативы к другой. Кроме того, есть и другие ограничения, касающиеся задач и обработки исключительных ситуаций. Эти ограничения будут рассмотрены в гл. 10 и 11.

**Пример.** Нижеследующие строки программы иллюстрируют некоторые из приведенных ограничений. Предполагается, что все переменные принадлежат к предопределенному типу INTEGER (Целый).

```
<<JOHN>>  if I = J
then
  <<MARY>> K := 2;
  case J - 3 is
    when 1 .. 5 => L := 5;
                    <<LOU>> K := L * 2;
    when 6 .. 10 => L := 10;
    -- Здесь нельзя размещать оператор
    -- GOTO LOU, поскольку запрещается переход
    -- от одной альтернативы оператора case
    -- к другой.
    when 0 | 13 => L := 1;
    when others => L := 0;
  end case;
else
  <<BOB>>  J := abs (I);
  if J = L
  then
    L := L * L; I := I + 1;
    -- Здесь можно поместить оператор GOTO BOB:
    -- он выполняет выход из внутреннего опера-
    -- тора IF. Оператор GOTO JOHN также будет
    -- правилен. Оператор вида GOTO MARY неве-
    -- рен, так как он производит переход от
    -- одной ветви оператора if к другой. Опера-
    -- тор GOTO LOU недопустим, поскольку он
    -- передает управление внутри составного
    -- оператора.
  end if;
end if;
```

Одной из причин, по которой в язык Ада включен оператор перехода, является необходимость перевода на этот язык программ, написанных на других языках программирования. Пользование оператором `goto`, однако, должно быть оправданным и его следует тщательно контролировать.

#### 4.1.4. Пустые операторы

В Аде есть *пустой оператор* (null statement), действие которого заключается в передаче управления следующему за ним оператору. Вот его формат:

```
null;
```

Одно из возможных мест применения пустого оператора — это операторы выбора. Вспомните, что условия выбора должны охватывать все возможные значения селектирующего выражения. Оператор `null` в этом случае служит для обозначения отсутствия выполнения каких-либо действий для некоторых альтернатив. Например, полагая, что `I` и `J` — целые, можно записать:

```
case I is
  when 1 => J := 1;
  when 3 => J := 2;
  when 5 | 8 => J := 3;
  when others => null;
end case;
```

#### 4.1.5. Операторы цикла, в которых не указаны условия повторения

В гл. 1 были рассмотрены циклы `while` (пока), а в гл. 2 — циклы `for` (для). Здесь вводится самая простая форма оператора цикла, представляющая собой *бесконечный цикл*. Она такова:

```
loop
-- Здесь располагаются какие-то операторы.
end loop;
```

В данном случае цикл будет выполняться бесконечно, если только какой-нибудь из операторов, расположенных внутри его, не передаст явно управление за пределы цикла. Среди прочих операторов языка Ада, которые могут передавать управление за пределы цикла, можно назвать оператор перехода `goto` и оператор выхода `exit`, который будет рассмотрен ниже.

#### 4.1.6. Именованные операторы цикла

Каждому оператору цикла можно присвоить некоторое имя. Это имя, если оно присутствует, нужно задавать в начале и в конце оператора цикла. Имя цикла — это идентификатор, за которым стоит двоеточие<sup>1)</sup>. Например, можно (опять-таки в предположении, что `I` и `J` — целые переменные) написать:

```
ONE_LOOP :— loop
-- Здесь располагаются операторы Ады.
end loop ONE_LOOP;
```

<sup>1)</sup> Не путать имя цикла с меткой! В отличие от метки на имя цикла нельзя выполнить переход по оператору `goto`. — Прим. перев.

Для цикла `while` можно записать:

```
TWO_LOOP: while I > 0
    loop
        J := J + I;
        I := I - 1;
    end loop TWO_LOOP;
```

Цикл `for` может быть следующим:

```
THREE_LOOP: for I in 3 .. 10 loop
    J := J + I;
end loop THREE_LOOP;
```

Имя цикла (например, `THREE_LOOP`) по аналогии с метками Ады объявляется неявно. Поэтому имя цикла, называемое также идентификатором цикла, считается объявленным в конце декларативной части самого внутреннего тела подпрограммы (задачи или пакета), содержащего это имя (см. гл. 7 и 10). Обратите внимание на то, что хотя и идентификатор цикла (например, `THREE_LOOP`), и параметр цикла (скажем, `I` из предыдущего примера) объявляются неявно, они создаются в разные моменты выполнения программы. Кроме того, параметр цикла известен только внутри оператора цикла.

### 4.1.7. Операторы выхода `exit`

*Оператор выхода* (`exit statement`) языка Ада применяется для организации завершения того цикла, в который он входит. Оператор выхода `exit` имеет следующие формы:

```
exit;
exit when условие;
```

Если циклы помечены, то оператор выхода принимает вид

```
exit имя_цикла;
exit имя_цикла when условие;
```

Условие, заданное в операторе `exit`, вычисляется, и если оно истинно, то осуществляется выход из цикла. Если условие отсутствует, то происходит безусловный выход из цикла.

Если имя цикла в операторе `exit` не указано, то происходит выход из самого внутреннего цикла, содержащего этот оператор. Если имеется несколько вложенных помеченных циклов, то можно задавать тот уровень, на который передает управление оператор `exit`, указывая в нем имя того цикла, выход за пределы которого необходимо произвести.

**Пример.** Следующие строки программы иллюстрируют использование операторов выхода. Будем считать, что все переменные — целые.

```
AA: for I in 1 .. 10
    loop
        case J is
            when 1 =>
                L := 0;
                BB: for K in 11 .. 20
                    loop
                        L := I * K + L;
                    exit when L > K * K;
```

```

-- Цикл с идентификатором BB вложен в цикл
-- с идентификатором AA. В операторе выход-
-- да имя цикла не указано, и поэтому ес-
-- ли будет истинно условие  $L > K * K$ , то
-- произойдет выход из цикла с именем BB,
-- а затем будет выполнен оператор
--  $L := L * L$ . Здесь используется оператор
-- выхода, расположенный в пределах цикла
-- for .
end loop BB;
when 5 | 8 =>
  L := I;
  CC : while L < 89
  loop
    exit AA when LL = 55;
    -- Цикл с именем CC вложен в цикл с
    -- идентификатором AA. В операторе
    -- выхода указано имя цикла AA. По-
    -- этому, если будет истинным усло-
    -- вие  $L = 55$ , то произойдет выход
    -- из цикла AA, а потом выполнится
    -- оператор  $K := L / 100$ . Если этот
    -- оператор выхода никогда не будет
    -- выполняться, а оператор цикла
    -- закончится, то после окончания
    -- цикла будет выполнен оператор
    --  $L := L * L$ .
    -- Это - пример оператора выхода,
    -- расположенного в пределах цикла
    -- while . Обратите внимание на
    -- то, что оператор выхода может
    -- располагаться в любом месте опе-
    -- ратора цикла.
    L := 2 * I + L;
  end loop CC;
  when others =>
    L := 0;
    DD : loop
      L := L + I + J;
      exit DD when L > 144;
      -- Обратите внимание на то, что
      -- было бы неправильным написать
      -- "exit DD when L > 144", так
      -- как цикл с именем DD не вло-
      -- жен в цикл BB. Это - пример
      -- оператора выхода, расположен-
      -- ного в пределах простейшего
      -- (бесконечного) оператора
      -- цикла.
    end loop;
  end case;
  L := L * L;
  exit when L > 8000;
end loop AA;
K := L / 100;

```

### 4.1.8. Операторы блока

В заключение введем еще один составной оператор — *оператор блока* (block statement). Полная форма этого оператора такова:

идентификатор\_блока:

`declare`

—Здесь даются объявления.

`begin`

—Здесь помещаются операторы.

`end` идентификатор\_блока;

Наличие идентификатора\_блока и декларативной части необязательно.

**Пример.** Блок без идентификатора:

`declare`

`I : INTEGER;`

`begin`

`I := J; J := K; K := I;`

`end;`

Блок без декларативной части:

`begin`

`I := 100;`

`end;`

Блок с идентификатором и декларативной частью:

`EXTRA:`

`declare`

`I : INTEGER := 100;`

`begin`

`if I > J then J := J + I; end if;`

`end EXTRA;`

Объекты, объявленные в данном блоке, (например, `I` в блоке с именем `EXTRA`), *локальны* по отношению к этому блоку, т.е. они неизвестны в остальной части подпрограммы, содержащей этот блок.

Блок выполняется последовательно. Вначале обрабатывается его декларативная часть. Термин «обрабатывается» относится к числу предпочтительных терминов языка Ада. Он обозначает те действия, которые выполняются над объявлениями. Дальнейшие подробности будут приведены в гл. 6. Затем выполняются операторы, входящие в блок. Память, требуемая для операторов, содержащихся в блоке, выделяется во время обработки его объявлений. Выделение памяти происходит каждый раз заново при выполнении блока. Таким образом, этот процесс может осуществляться многократно.

Одна из сфер применения операторов блока связана с обработкой исключительных ситуаций (см. гл. 11).

### 4.1.9. Программа, в которой используются оператор выбора `case` и оператор выхода `exit`

Завершим этот раздел примером программы, в которой употребляются операторы `case` и `exit`. Здесь используется также еще одна подпрограмма из пакета `TEXT_IO` — процедура `GET_LINE`, которая считывает входную строку, содержащую данные, присваивает ее переменной типа `STRING` и подсчитывает количество прочитанных символов. Таким образом, процедура `GET_LINE` имеет два аргумента: имя строки и имя целой переменной — счетчика введенных символов.

В каждой входной записи размещаются фамилия и номер телефона. Вначале идет фамилия, а потом (через запятую) — номер телефона. Номера телефонов состоят из десяти цифр и/или букв и могут иметь различный формат. Они могут состоять целиком из десяти цифр и включают телефонный код местности, например 2125551212. В обозначения номеров могут быть вставлены дефисы, например 212-555-1212. Вместо некоторых цифр могут стоять буквы, например 212jkl1A1B, что эквивалентно 2125551212<sup>1)</sup>.

Фамилии<sup>2)</sup> могут быть представлены двумя способами. В первом формате за первым именем следуют по крайней мере один пробел, затем необязательный инициал, соответствующий среднему имени, и точка. Далее идет по меньшей мере один пробел и затем последнее имя. Во втором формате вначале располагается последнее имя, за ним запятая, затем пробел, потом необязательный инициал среднего имени с точкой и в конце первое имя (после пробела). Первое и последнее имена и инициал среднего имени состоят из букв обоих регистров. В именах могут встречаться апострофы и тире, например Robbe-Grillet и O'Hara.

Вот — примеры входных строк:

```
John S. Burundi, 213-9AB-COOL
Christensen, Paul, 3129129292
Kandisky, F. Richard, 914-abcabcd
```

Признаком конца данных служит строка, содержащая только символы XX.

Выводятся строки с правильными фамилиями абонентов и номерами их телефонов. При этом печатаются только первое и последнее имена (буквами верхнего регистра) и телефонный номер с использованием только цифр и двух дефисов. Первое и второе имена и номер телефона отделяются друг от друга одним пробелом. Если в строке неверно записаны номер телефона или фамилия (например, если в номере слишком мало цифр или в имена входят специальные символы), то входную строку следует распечатать без изменений, сопроводив ее сообщением вида «неверный номер или фамилия».

Вот пример выходной информации с использованием приведенных выше фамилий:

```
JOHN BURUNDI 213-922-2665
RICHARD KANDISKY 914-222-2223
```

Связь букв и цифр в номерах телефонов следующая: с каждой цифрой (от 2 до 9 включительно) ассоциируется группа латинских букв. С цифрой 2, например, связаны буквы ABC, а с цифрой 9 — WXY.

### Программа NAME\_PHONE

```
with TEXT_IO, use TEXT_IO;
procedure NAME_PHONE is
  LINE_LN : NATURAL;
  -- Вспомните, что NATURAL — это предопределен-
  -- ный подтип, имеющий в качестве исходного
  -- тип INTEGER. Значения этого подтипа лежат
  -- в пределах от 0 до INTEGER'LAST.
  F_NAME_LN, L_NAME_LN, PHONE_LN : NATURAL;
  -- Эти переменные несут информацию о
```

<sup>1)</sup> В США буквы в этих обозначениях служат эквивалентом цифр. — Прим. перев.

<sup>2)</sup> В переводе употребляется слово «фамилия» для полной идентификации человека. В США у человека может быть «первое имя» (аналог нашего имени), так называемые «среднее имя» и «последнее имя» (аналог нашей фамилии). — Прим. перев.

```

-- количество символов в первом имени,
-- среднем инициала, фамилии и номере
-- телефона.
INP_LINE : STRING(1 .. LINE_LN);
-- Вспомните, что, если LINE_LN := 0, то
-- INP_LINE станет массивом с пустым
-- диапазоном индексов.
WRK_LINE : STRING(1 .. LINE_LN);
-- Эта строка используется при внесе-
-- нии изменений в исходную строку.
F_NAME : STRING(1 .. F_NAME_LN);
-- Здесь будет записано первое имя,
-- если только оно правильно.
L_NAME : STRING(1 .. L_NAME_LN);
-- Сюда будут помещены фамилии.
PHONE_NO : STRING(1 .. PHONE_LN);
-- Здесь будут размещаться телефонные
-- номера.
NO_COMMAS : INTEGER;
-- Значение этой переменной равно
-- количеству запятых в строке.
BAD_DATA : BOOLEAN;
-- Эта переменная будет иметь значение
-- TRUE, если в данной строке представлены
-- неправильные данные.
STRT_POS, END_POS : NATURAL ;
-- Эти переменные содержат номер началь-
-- ной и конечной позиции имен или номе-
-- ров телефонов.
DIGIT_CT, CURR_DIG : INTEGER;
-- Это - счетчики. Во всех процедурах,
-- приведенных далее, используется пере-
-- менная WRK_LINE.
procedure LOW_TO_UPPER_N_CT_COMMAS is
begin
-- Преобразуем каждую букву нижнего
-- регистра в букву верхнего регистра и
-- подсчитаем количество запятых. Одна
-- запятая означает, что вначале распо-
-- лагается первое имя; две запятые -
-- то, что вначале располагается фамилия;
-- иное количество запятых свидетельству-
-- ет об ошибке.
NO_COMMAS := 0;
for I in 1 .. LINE_LN
loop
case WRK_LINE(I) is
when 'a' .. 'z' =>
WRK_LINE(I) := CHARACTER'VAL(
CHARACTER'POS('A') -
CHARACTER'POS('a') +
CHARACTER'POS(WRK_LINE(I) ));
-- Здесь выполняется преобразование
-- буквы от нижнего регистра к верх-
-- нему. Разница в относительной по-
-- зиции 'A' и 'a' будет и разницей
-- между положением любой другой

```

```

-- буквы нижнего регистра и соответ-
-- ствующей ей буквой верхнего реги-
-- стра. Вспомните, что строки - это
-- массивы предопределенного перечи-
-- сляемого типа CHARACTER.
when ',' => NO_COMMAS := NO_COMMAS+1;
when others => NULL;
end case;
end loop;
end LOW_TO_UPPER_N_CT_COMMAS;
procedure IGNORE_LEADING_SPACES is
-- Эта процедура устанавливает значение
-- переменной STRT_POS, равное номеру
-- позиции в строке первого символа,
-- отличающегося от пробела.
begin
  for I in 1 .. LINE_LN
    loop
      STRT_POS := I;
      exit when WRK_LINE(I) /= ' ';
    end loop;
end IGNORE_LEADING_SPACES;
procedure FIND_NEXT_SP_OR_COMMA is
-- Эта процедура устанавливает значение
-- переменной END_POS, равное номеру
-- позиции последнего символа, не равно-
-- го пробелу или запятой и расположен-
-- ного после STRT_POS.
begin
  END_POS := STRT_POS;
  for I in STRT_POS .. LINE_LN
    loop
      exit when WRK_LINE(I) = ' ' or
        WRK_LINE(I) = ',';
      END_POS := I;
    end loop;
end FIND_NEXT_SP_OR_COMMA;
procedure PLACE_SPACES is
-- Эта процедура заменяет символы, распо-
-- ложенные между STRT_POS и END_POS, на
-- пробелы и заменяет первую встреченную
-- запятую на пробел. Она останавливает-
-- ся после первого встреченного символа,
-- не равного пробелу и расположенного
-- после END_POS.
begin
  for I in STRT_POS .. END_POS
    loop
      WRK_LINE(I) := ' ';
    end loop;
  for I in END_POS + 1 .. LINE_LN
    loop
      if WRK_LINE(I) = ','
        then
          WRK_LINE(I) := ' ';
          exit;
        end if;
      exit when WRK_LINE(I) /= ' ';
    end loop;
  end loop;
end PLACE_SPACES;

```



```

end loop;
end PLACE_SPACES;
procedure IS_CORRECT_NAME is
-- Эта процедура проверяет то, что
-- апострофы или черточки, входящие в
-- состав имени, окружены буквами.
-- Если это не так, то переменная
-- BAD_DATA получает значение TRUE.
begin
  for I in STRT_POS .. END_POS
  loop
    case WRK_LINE(I) is
      when 'A' .. 'Z' => NULL
      when '-' | "'" =>
        if I = STRT_POS or I = END_POS
          -- Разрешается наличие в имени
          -- черточек и апострофов, но
          -- только окруженных другими
          -- символами.
        then
          BAD_DATA := TRUE;
          exit;
          -- Эта строка - оператор
          -- безусловного выхода.
        end if;
      when others => BAD_DATA := TRUE;
      exit;
    end case;
  end loop;
end IS_CORRECT_NAME;
procedure XTR_N_VAL_FIRST_NAME is
-- Эта процедура проверяет первое имя, выделяя-
-- ет его и записывает в F_NAME. Если в имени
-- обнаружен неверный символ, то переменная
-- BAD_DATA получает значение TRUE.
begin
  IGNORE_LEADING_SPACES;
  FIND_NEXT_SP_OR_COMMA;
  IS_CORRECT_NAME;
  L_NAME_LN := END_POS - STRT_POS + 1;
  L_NAME := WRK_LINE ( STRT_POS .. END_POS );
  -- Это - оператор присваивания вырезки.
  PLACE_SPACES;
end XTR_N_VAL_FIRST_NAME;
procedure XTR_N_VAL_MDL_INIT is
-- Эта процедура ищет средний инициал,
-- за которым следуют точка и пробел или
-- запятая. Если инициал найден, то эти
-- символы заменяются пробелами; в про-
-- тивном случае переменная WRK_LINE
-- не изменится.
begin
  IGNORE_LEADING_SPACES;
  if WRK_LINE (STRT_POS) in 'A' .. 'Z' and
    WRK_LINE (STRT_POS + 1) = '.' and
    (WRK_LINE (STRT_POS + 2) = ',' or
     WRK_LINE (STRT_POS + 2) = ' ')

```

```

    then
      FIND_NEXT_SP_OR_COMMA;
      PLACE_SPACES;
    end if;
end XTR_N_VAL_MDL_INIT;
procedure XTR_N_VAL_LAST_NAME is
  -- Эта процедура выполняет действия, сходные
  -- с действиями процедуры XTR_N_VAL_FIRST_NAME,
  -- за исключением того, что она выделяет не
  -- первое имя, а фамилию.
begin
  IGNORE_LEADING_SPACES;
  FIND_NEXT_SP_OR_COMMA;
  IS_CORRECT_NAME;
  F_NAME_LN := END_POS - STRT_POS + 1;
  F_NAME := WRK_LINE ( STRT_POS .. END_POS );
  -- Здесь используется присваивание вырезки.
  PLACE_SPACES;
end XTR_N_VAL_LAST_NAME;
procedure XTR_N_VAL_PHONE is
  -- Эта процедура проверяет номер телефона.
  -- Если номер правильный, то он преобразует-
  -- ся в цифровой формат и запоминается в
  -- PHONE_NO.
begin
  IGNORE_LEADING_SPACES;
  FIND_NEXT_SP_OR_COMMA;
  DIGIT_CT := 0;
  for I in STRT_POS .. END_POS
    loop
      case WRK_LINE (I) is
        when '0' .. '9' => DIGIT_CT := DIGIT_CT+1;
        when '-' => if I = STRT_POS or
                     I = END_POS
          -- Черточки разрешены, но только в окруже-
          -- нии других символов.
          then
            BAD_DATA := TRUE;
            exit;
            -- Это - безусловный выход
            -- из цикла.
          end if;
        when 'A' .. 'C' => WRK_LINE (I) := '2' ;
        when 'D' .. 'F' => WRK_LINE (I) := '3' ;
        when 'G' .. 'I' => WRK_LINE (I) := '4' ;
        when 'J' .. 'L' => WRK_LINE (I) := '5' ;
        when 'M' .. 'O' => WRK_LINE (I) := '6' ;
        when 'P' | 'R' | 'S' => WRK_LINE (I) := '7' ;
        when 'T' .. 'V' => WRK_LINE (I) := '8' ;
        when 'W' .. 'Y' => WRK_LINE (I) := '9' ;
        when others      => BAD_DATA := TRUE;
      end case;
    loop
      exit;
    end loop;
  end for;
  if not BAD_DATA
    then
      DIGIT_CT := DIGIT_CT + 1;
    end if;

```

```

end loop;
if DIGIT_CT /= 10
then
  BAD_DATA := TRUE;
end if;
if not BAD_DATA
then
  PHONE_LN := 12;
  CURR_DIG := 3;
  for I in STRT_POS .. END_POS
  loop
    if WRK_LINE (I) in '0' .. '9'
    then
      PHONE_NO ( CURR_DIG ) := WRK_LINE (I);
      CURR_DIG := CURR_DIG + 1;
    end if;
  end loop;
  PHONE_NO (1 .. 3) := PHONE_NO ( 3 .. 5);
  -- Это - опять присваивание вырезки.
  PHONE_NO (4) := '-';
  PHONE_NO (5 .. 7) := PHONE_NO (6 .. 8);
  PHONE_NO(8) := '-';
end if;
PLACE_SPACES;
end XTR_N_VAL_PHONE;
procedure DISP_ERROR is
-- Процедура выдает сообщение об ошибке и
-- отображает строку с неверными данными.
begin
  PUT (" This line is invalid: ");
  PUT( INP_LINE );
  NEW_LINE;
end DISP_ERROR;
begin
  GET_LINE (INP_LINE, LINE_LN) ;
  -- Теперь считывается строка с фамилией або-
  -- нента и номером телефона. Далее расположен
  -- простейший цикл.
  while INP_LINE /= "XX"
  loop
    WRK_LINE := INP_LINE;
    LOW_TO_UPPER_N_CT_COMMAS;
    BAD_DATA := FALSE;
    case NO_COMMAS is
      when 1 =>
        -- Эта альтернатива выбирается, если за
        -- первым именем следует фамилия.
        XTR_N_VAL_LAST_NAME;
        XTR_N_VAL_FIRST_NAME;
        XTR_N_VAL_MDL_INIT;
        XTR_N_VAL_PHONE;
      when 2 =>
        -- Эта альтернатива выбирается, если за
        -- фамилией стоит запятая, а затем -
        -- первое имя.
        XTR_N_VAL_FIRST_NAME;
        XTR_N_VAL_MDL_INIT;

```

```

XTR_N_VAL_LAST_NAME;
XTR_N_VAL_PHONE;
when others => BAD_DATA := TRUE;
end case;
if BAD_DATA
then
  DISP_ERROR;
else
  PUT (F_NAME); PUT ( ' ');
  PUT (L_NAME); PUT ( ' ');
  PUT (PHONE_NO);
  NEW_LINE;
end if;
GET_LINE ( INP_LINE, LINE_LN);
end loop;
end NAME_PHONE;

```

## 4.2. ИМЕНА, ЗНАЧЕНИЯ И ВЫРАЖЕНИЯ

В этом разделе рассматриваются новые аспекты понятий, широко использовавшихся в предыдущих главах. Обсуждаются имена, значения и выражения.

### 4.2.1. Имена

*Имена* широко применялись в предыдущих главах в качестве идентификаторов переменных, констант, типов, подтипов и подпрограмм. В данной главе были введены новые виды идентификаторов: *метки*, *имена блоков* и *имена циклов*. Завершают список возможных идентификаторов языка Ада имена *задач* и их входов, а также названия исключительных ситуаций (см. гл. 10 и 11).

Кроме идентификаторов, в Аде есть и другие виды имен, например, *атрибуты*, *составные имена* (selected components), *вырезки* из массивов, *индексированные компоненты* (последние три вида имен были введены в гл. 2), *символьные литералы* и *обозначения операций*. Здесь кратко рассмотрим особенности составных имен и индексированных компонент.

*Индексированные компоненты* использовались в гл. 2 для обозначения элементов массива. Как будет видно из материала гл. 10, они также могут применяться для обозначения входа в семействе входов. Общая форма индексированной компоненты такова: *имя\_или\_обращение\_к\_функции* (одно\_или\_более\_выражений). Если выражений несколько, то они отделяются друг от друга запятыми. Совокупность этих выражений задает определенное значение компоненты. Использование вызовов функций в индексированных компонентах будет рассмотрено в следующей главе.

Следует отметить, что компоненты многомерных массивов и компоненты массивов, состоящих из массивов, — это разные понятия, имеющие к тому же и различные формы записи. Пусть, например, действуют объявления:

```

type X is array (1 .. 10, 1 .. 10 )
  of INTEGER;
type Y is array ( 1 .. 10 ) of INTEGER;
type YY is array ( 1 .. 10 ) of Y;
A : X;
B : YY;

```

Здесь А – двумерный массив. Для обращения к одной из его компонент следует указать два значения или в общем случае два выражения, например А (2,5). Однако массив В – это массив, состоящий из массивов. Здесь В (2) (5) обозначает пятую компоненту массива В (2).

Обратимся теперь к *составным именам*. Их общая форма такова:

имя\_или\_обращение\_к\_функции.селектор

Селектором может служить идентификатор, символьный литерал, обозначение операции (см. гл. 5) или зарезервированное слово all (все).

Мы уже употребляли составные имена для обозначения компонент структуры (см. гл. 2) или для объектов, на которые указывали ссылочные переменные (см. гл. 3). Завершают список возможных составных имен ресурсы, объявленные в видимой части пакетов (см. гл. 7), входы задач (см. гл. 10) и ресурсы, объявленные в охватывающем данный участок программы теле подпрограммы, теле пакета, теле задачи, а также в охватывающем блоке или цикле.

**Пример.** Следующие строки программы демонстрируют использование составных имен в охватывающем блоке.

```
AA : declare
  I : INTEGER;
begin
  I := 10;
  BB : declare
    I, J : INTEGER;
  begin
    J := AA.I; BB.I := J ** 2;
    AA.I := J mod 5 + BB.I / 5;
  end BB;
end AA;
```

Здесь AA.I и BB.I – два составных имени. Членом «имя», т. е. префиксом, здесь является идентификатор блока (AA или BB), а членом «селектор» – идентификатор I, обозначающий имя переменной. Форма записи AA.I и BB.I внутри блока BB необходима для того, чтобы указать, какую именно переменную с именем I мы хотим использовать: ту, которая объявлена в блоке AA, или ту, которая объявлена в блоке BB. В этом случае вступают в силу так называемые правила видимости, которые будут изложены в гл. 6 и 7.

## 4.2.2. Значения

Обратимся теперь к значениям в языке Ада. Вспомните, что тип в Аде можно определить как совокупность значений и операций над ними. *Величины* и *значения* Ады могут принадлежать к типам, не имеющим компонент, например к перечисляемым, целым и действительным типам. Они могут относиться и к типам, имеющим компоненты, например к комбинированным типам (т. е. структурам) и к регулярным типам (т. е. массивам).

Перечисляемые, целые и действительные типы объединяются в группу скалярных типов. Как можно догадаться, значения, принадлежащие к ним, называются *скаляр-*

ными значениями. Значения, относящиеся к составным типам, т.е. к типам, объекты которых имеют компоненты, называются *составными значениями*, или *агрегатами*. Первые они были введены в гл. 2 для регулярных и комбинированных типов. При этом использовалась разновидность агрегатов, называемая *позиционным агрегатом*.

Литералы мы употребляли для представления скалярных значений. Это были числовые и перечисляемые литералы. К последним относятся и символьные литералы. Литералы также использовались и для описания составных значений (например, символьных строк), и для обозначения ссылочных значений (null). Числовые литералы относятся к типу `универсальный_целый` или `универсальный_действительный`. Числовые литералы имеют важную отличительную особенность. Они принадлежат к типам, имеющим произвольную точность. Поэтому числовые литералы являются предпочтительным способом введения констант как величин, принадлежащих к вполне конкретному (т.е. не универсальному) типу.

Теперь введем новый вид агрегатов для составных значений — *агрегаты с поименованными компонентами*. В таких агрегатах каждому значению компоненты предшествует ее имя или значение ее индекса, за которым ставится составной символ `=>`. Преимущество такой формы записи заключается в том, что значения компонент можно записывать в произвольном порядке.

**Пример.** Пусть имеются объявления:

```
type WEIGHT_CLASS is array (1 .. 5) of FLOAT;
type EMPL_REC is
  record
    F_NAME : STRING (1 .. 10);
    SOS_ID : STRING (1 .. 9);
    SALARY : FLOAT;
  end record;
CURR_EMPL : EMPL_REC;
WEIGHT_TABLE : WEIGHT_CLASS;
```

Тогда агрегаты можно записать несколькими способами. Вот один из них:

```
("NICHOL      ", "123456789", 550.25)
```

Это позиционный агрегат. Другой способ записи:

```
(F_NAME => "NICHOL      ",
 SOS_ID  => "123456789",
 SALARY  => 550.25)
```

Этот агрегат с поименованными компонентами. Третий способ:

```
(SALARY => 550.25, F_NAME => "NICHOL      ",
 SOS_ID  => "123456789")
```

Это тот же агрегат, но порядок следования именованных компонент выбран произвольно. Вот пример корректной записи оператора присваивания:

```
CURR_EMPL :=
(SALARY => 550.25, F_NAME => "NICHOL      ",
 SOS_ID  => "123456789");
```

Для переменной `WEIGHT_TABLE` можно представить два эквивалентных агрегата. Позиционный агрегат записывается так:

(1.0, 3.0, 5.0, 8.0, 13.0)

В форме с поименованными компонентами его можно записать так:

(1 => 1.0, 2 => 3.0, 3 => 5.0, 4 => 8.0, 5 => 13.0)

или так:

(3 => 5.0, 1 => 1.0, 2 => 3.0, 4 => 8.0, 5 => 13.0)

При записи агрегата можно использовать диапазон значений индексов, например, так:

(1..5 => 3.5)

что эквивалентно следующей записи:

(3.5, 3.5, 3.5, 3.5, 3.5)

Более того, разрешается употреблять зарезервированное слово `others` (другие) для указания оставшихся значений, имея в виду компоненты, не упомянутые ранее. Например, можно записать:

(2..4 => 0.5, others => 1.0)

что эквивалентно записи

(1.0, 0.5, 0.5, 0.5, 1.0)

Слово `others` должно быть последним среди имен компонент агрегата.

Можно присваивать значения нескольким (не обязательно следующим подряд) компонентам, если использовать символ `|`. Этот символ разделяет имена компонент, получающих одно и то же значение. Например:

(1 | 3 | 5 => 3.0, others => 5.0)

—это корректный агрегат с поименованными компонентами. Он эквивалентен записи

(3.0, 5.0, 3.0, 5.0, 3.0)

В языке Ада разрешено использование *неоднозначных агрегатов*, т.е. агрегатов, которые могут принадлежать сразу к нескольким различным типам. Например, если кроме типа `WEIGHT_CLASS`, будет объявлен еще и тип `DIST_CLASS`:

type `DIST_CLASS` is array (0 .. 7) of `FLOAT`;

и имеется агрегат вида:

(1 => 13.0, 3 | 5 => 8.0, others => 0.0)

то этот агрегат может относиться как к типу `DIST_CLASS`, так и к типу `WEIGHT_CLASS`. Неоднозначность будет устранена, если агрегат (или любое другое выражение при аналогичной ситуации) уточнить («квалифицировать») при помощи имени типа. В этом случае следует записать так:

`DIST_CLASS'` (1 => 13.0, 3 | 5 => 8.0, others => 0.0)

если мы обращаемся к значению типа `DIST_CLASS`. Если же нужно обратиться к значению типа `WEIGHT_CLASS`, то нужно написать так:

`WEIGHT_CLASS'` (1 => 13.0, 3 | 5 => 8.0, others => 0.0)

Формат записи квалификатора значения агрегата чем-то схож с использованием атрибутов: за именем типа или подтипа следует апостроф, за которым располагается само значение агрегата<sup>1)</sup>. В разд. 6.1.2 будут даны более подробные сведения об устранении неоднозначности величин и о квалификаторах.

### 4.2.3. Выражения

Теперь перенесем наше внимание на выражения языка Ада. Как упоминалось в гл. 1, наиболее элементарным видом выражения является *простейшее выражение* (primary). Агрегат — это простейшее выражение. Такими же выражениями являются и уточненные величины из приведенных выше примеров. Другие простейшие выражения, которые уже встречались, — это числовые литералы, символьные строки, пустое значение “null”, имена и генераторы. Кроме них, в этот список также входят обращения к функциям, о которых будет рассказано в следующей главе, преобразования типов и вообще любые выражения, заключенные в скобки. Простейшие выражения служат основой для построения более сложных выражений.

В нижеследующих определениях и примерах будем считать, что все переменные относятся к целому типу.

*Множитель* — это либо любое простейшее выражение, возведенное в степень (например,  $I ** J$ ), либо абсолютное значение простейшего выражения ( $abs\ I$ ), либо простейшее выражение, перед которым стоит зарезервированное слово *not* (не) (например,  $not\ 0$ ).

*Терм* — это любой множитель, за которым следует обозначение одной из операций  $*$ ,  $/$ ,  $mod$ ,  $rem$ , а затем — другой множитель. Пример терма:  $I * J$ .

*Простое выражение* — это терм, перед которым может стоять знак  $(+)$  или  $(-)$ , или терм, за которым следует символ операции  $+$ ,  $-$  или  $\&$ , а затем — другой терм. Пример простого выражения:  $I + J$ .

*Отношение* — это либо простое выражение; либо простое выражение, за которым следует обозначение одной из операций сравнения  $=$ ,  $/=$ ,  $<$ ,  $<=$ ,  $>$ ,  $>=$ , а затем — другое простое выражение (пример:  $I < J$ ); либо простое выражение, за которым следует зарезервированное слово *in* или *not in*, а далее — диапазон значений, тип или подтип (пример:  $I\ in\ 1..7$ ).

*Выражение* — это либо отношение; либо отношение, за которым следует символ логической операции *or* (или), *and* (и) или *xor* (исключающее или), а затем — другое отношение (пример:  $I < J\ and\ J > K$ ); либо отношение, за которым следуют ключевые слова *and then* (и тогда), а далее — еще одно отношение (пример:  $I < J\ and\ then\ I = 0$ ); либо отношение, за которым следуют зарезервированные слова *or else* (или когда), а затем располагается другое отношение (пример:  $I = J\ or\ else\ J = 3$ ). Операция *and then* называется *сокращенной конъюнкцией*, а операция *or else* — *сокращенной (инклюзивной) дизъюнкцией*. Ниже мы детально рассмотрим их.

**Пример.** Вот некоторые примеры, цель которых — помочь разобраться в иерархической структуре понятий простейшее выражение, множитель, терм, простое выражение, отношение и выражение.

Простейшее выражение:  $(I ** J * K + L > M\ or\ N = P)$

Выражение (но не отношение):  $I ** J * K + L > M\ or\ N = P$

Отношение (но не простое выражение):  $I ** J * K + L > M$

Простое выражение (но не терм):  $I ** J * K + L$

Терм (но не множитель):  $I ** J * K$

Множитель (но не простейшее выражение):  $I ** J$

<sup>1)</sup> Для атрибутов порядок обратный: сначала идет величина, а затем апостроф и атрибут. — Прим. перев.



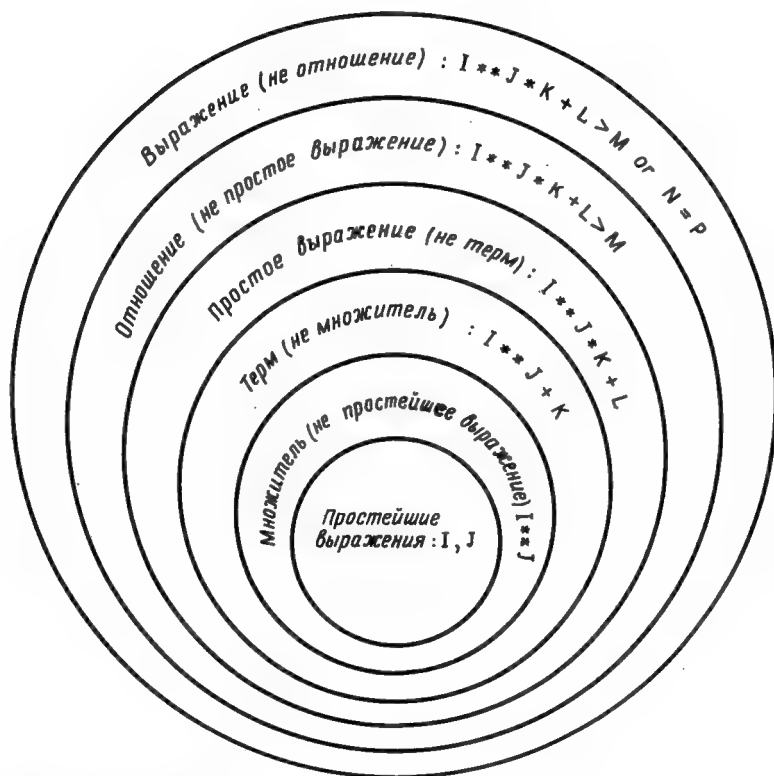


Рис. 4.1. Иерархия выражений.

Рис. 4.1 иллюстрирует эту иерархию.

Логическая операция хог (исключающее или) ранее нами не использовалась. Результат вычисления выражения, в состав которого входит эта операция, получается следующим образом. Если  $I$  и  $J$  — логические переменные, то выражение  $I$  хог  $J$  дает значение FALSE, когда  $I$  и  $J$  имеют одинаковое значение. В противном случае значением выражения будет TRUE.

Мы упомянули *сокращенные формы* логических операций and и or — операции and then и or else. Эти сокращенные формы предписывают транслятору формировать объектный код таким образом, чтобы вначале вычислялась левая часть выражения. Правая часть будет вычисляться только в том случае, если одна левая не определит уже значение всего выражения.

**Пример.** Он иллюстрирует выгоду применения сокращенных форм логических операций. Здесь приведена модифицированная процедура FIND\_NEXT\_SP\_OR\_COMMA из программы PHONE\_NAME, в которой теперь употребляется сокращенная логическая операция and then.

```

procedure FIND_NEXT_SP_OR_COMMA is
  I : INTEGER;
begin
  END_POS := STRT_POS;
  I := STRT_POS;

```

```

while I <= LINE_LN and then
  WRK_LINE(I) /= ' ' and then
    WRK_LINE(I) /= ','
  loop
    I := I+1;
  end loop;
END_POS := I - I / (LINE_LN + 1);
-- Член I/(LINE_LN+1) нужен для того, чтобы
-- вычесть 1 из I, если последнее значение I
-- равняется LINE_LN+1.
and FIND_NEXT_SP_OR_COMMA;

```

### Выражение

$I \leq \text{LINE\_LN}$  and then  $\text{WRK\_LINE}(I) \neq ' '$  and then  $\text{WRK\_LINE}(I) \neq ','$

вычисляется следующим образом. Вначале вычисляется отношение  $I = \text{LINE\_LN}$ . И только в том случае, если оно истинно, будет вычисляться отношение  $\text{WRK\_LINE}(I) \neq ' '$ . Только если и в этом случае результат окажется TRUE, то будет вычислено последнее отношение  $\text{WRK\_LINE}(I) \neq ','$ .

Заметьте, что если здесь заменить сокращенную логическую операцию and then на обычную логическую операцию and, то индекс массива мог бы выйти за пределы предписанного диапазона. Так, если бы  $I$  стало равно  $\text{LINE\_LN} + 1$ , то соответствующая ему компонента  $\text{WRK\_LINE}(\text{LINE\_LN} + 1)$  отсутствовала бы в массиве.

За исключением сокращенных форм логических операций, не предполагается какой-либо обязательный порядок, в котором должны предварительно вычисляться значения обоих операндов для выполнения операции. Вспомните, что у операции and более высокий приоритет, чем у операции or. Приоритет операции and then такой же, как у операции and, а приоритет or else и хог такой же, как у or.

Логические операции можно применять по отношению к одномерным массивам, элементы которых принадлежат к логическому типу. При этом операции выполняются с каждым элементом отдельно. Пусть, например, имеются такие объявления:

```

type SWITCHES is array (1 .. 7) of BOOLEAN;
CURR_STATUS, PREV_STATUS : SWITCHES;

```

Тогда можно написать следующие операторы:

```

PREV_STATUS := (1 .. 4 | 7 => TRUE, others => FALSE);
CURR_STATUS := PREV_STATUS xor PREV_STATUS;

```

Они эквивалентны операторам:

```

CURR_STATUS := (1 .. 7 => FALSE);
PREV_STATUS := (5 .. 6 => FALSE, others => TRUE);

```

Можно также записать:

```

CURR_STATUS := CURR_STATUS or PREV_STATUS;

```

Это эквивалентно следующему оператору:

```

CURR_STATUS := (1 .. 4 | 7 => TRUE, others => FALSE);

```

В табл. 4.1 приведена сводка операций языка Ада, типов операндов, к которым они применяются, и типов результатов этих операций.

Таблица 4.1. Операции языка Ада

Обозначение операции	Название операции	Типы операндов		Тип результата
		Левый операнд	Правый операнд	
Унарные операции				
+	Знак плюс	Числовой		Тот же числовой тип, что и у операнда
-	Знак минус	»		То же
abs	Абсолютное значение	»		»
not	Логическое отрицание	Логический		»
Бинарные операции				
+	Сложение	Числовой	Тот же числовой тип, что и у левого операнда	Тот же числовой тип, что и у операндов
-	Вычитание	»	Тот же числовой тип, что и у левого операнда	»
&	Сцепление	Одномерный регулярный тип	Тот же одномерный регулярный тип, что и у левого операнда	Тот же одномерный регулярный тип, что и у операндов
		»	Тип компонент левого операнда	Тот же одномерный регулярный тип, что и у левого операнда
		Тип компонент, образующих регулярный тип	Регулярный одномерный тип с компонентами типа левого операнда	Тот же одномерный регулярный тип, что и у правого операнда
		Тип элементов	Тип элементов	Любой регулярный тип
mod	Остаток от деления	Целый	Целый	Целый
rem	Остаток от деления	»	»	»
*	Умножение	»	»	»
		Плавающий	Плавающий	Плавающий
		Фиксированный	Целый	Тот же фиксированный тип,

		Целый	Фиксированный	что и у левого операнда
		Фиксированный	Фиксированный	Тот же фиксированный тип, что и у правого операнда
/	Деление	Целый Плавающий Фиксированный	Целый Плавающий Целый	Целый Плавающий Тот же фиксированный тип, что и у левого операнда
		»	Фиксированный	Универсальный фиксиро- ванный тип
**	Возведение в степень	Целый	Натуральный	Тот же числовой тип, что и у левого операнда
		Плавающий	Целый	Тот же плавающий тип, что и у левого операнда
=	Равенство	Скалярный Составной Ссылочный	Тип левого операнда То же	Логический То же
/=	Неравенство	Скалярный Составной Ссылочный	» » »	» » »
<, <=	Меньше, меньше или равно	Скалярный	»	»
>, >=	Больше, больше или равно	Дискретный регулярный тип	»	»
in, not in	Операции проверки при- надлежности	Значение, принадлежащее к типу правого операнда	Диапазон, тип или подтип	»
and	Конъюнкция	Логический	Логический	»
or xor	Инклюзивная дизъюнкция Эксклюзивная дизъюнкция	Логический регулярный	Тот же логический регуляр- ный тип, что и у левого операнда	Тот же логический регуляр- ный тип, что и у операн- дов
and then	Сокращенная форма конъ- юнкции	Логический	Логический	Логический
or else	Сокращенная форма инклю- зивной дизъюнкции	»	»	»

### 4.3. КОМБИНИРОВАННЫЕ ТИПЫ С ВАРИАНТАМИ

*Вариантные комбинированные типы*—это особый вид комбинированных типов с дискриминантами. Они дают возможность программисту указывать один или несколько альтернативных списков компонент, которые должны быть включены в объект этого типа (т.е. в структуру). Вариантная часть комбинированного типа, если она присутствует, должна располагаться непосредственно перед ключевыми словами `end record`. Она имеет форму:

```

case имя_дискриминанта is
    when одно_или_несколько_условий_выбора => список_компонент
    -- Здесь могут присутствовать несколько таких вариантов.
end case;
```

Если после зарезервированного слова `when` располагаются несколько условий выбора, то они отделяются друг от друга вертикальной чертой `|`.

В качестве условия выбора можно употреблять статическое простое выражение, дискретный диапазон, зарезервированное слово `others` или простое имя компоненты, под которым подразумевается только идентификатор компоненты (скажем, без индексов). Зарезервированное слово `others` может появляться как условие выбора только для последней альтернативы. Поскольку переменные комбинированного типа (т.е. структуры) могут служить компонентами других комбинированных типов, образуя так называемые вложенные комбинированные типы, то могут получаться и вложенные вариантные части. Условия выбора должны охватывать все множество возможных значений дискриминанта. Наличие совпадающих значений в разных условиях выбора недопустимо.

Список компонент—это список компонент комбинированного типа, т.е. последовательность объявлений компонент структуры. Если список компонент пуст, то должна присутствовать компонента `null` (пусто).

**Пример.** Следующие строки программы демонстрируют объявления комбинированного типа с вариантной частью.

```

type LABOR is (SALARIED, HOURLY, SUBCONTRACTOR);
type EMPLOYEE (LAB_KIND : LABOR := SALARIED) is
    -- Вспомните (см. гл. 2), что LAB_KIND : LABOR
    -- - это дискриминантная часть. Она имеет на-
    -- чальное значение, которое будет принято по
    -- умолчанию. Имя дискриминанта - LAB_KIND.
    record
        F_NAME : STRING ( 1 .. 10 );
        L_NAME : STRING ( 1 .. 15 );
        SOS_ID : STRING ( 1 .. 9 );
        case LAB_KIND is
            when SALARIED => YEARLY_SAL : FLOAT;
                               Y_T_DATE_TOTAL : FLOAT;
            when HOURLY    => HOURLY_PAY : FLOAT;
                               HOURS_WORKED : FLOAT;
                               OVERTIME_HRS : FLOAT;
            when SUBCONTRACTOR => DATLY_FEE : FLOAT;
                               Y_T_DATE_DAYS : NATURAL;
        end case;
    end record;
```

Теперь приведем объявления переменных типа EMPLOYEE:

```
SAL_EMPLOYEE : EMPLOYEE ( SALARIED );
CURR_EMPLOYEE : EMPLOYEE;
-- Это об'явление - правильное, так как при
-- об'явлении комбинированного типа
-- EMPLOYEE было задано начальное значения
-- дискриминанта, которое принимается
-- по умолчанию.
CLERICAL_EMPL : EMPLOYEE ( HOURLY );
```

Далее дадим два примера присваивания агрегата, относящегося к типу EMPLOYEE:

```
CURR_EMPLOYEE := (LAB_KIND => SALARIED, F_NAME =>
"ROBERT", L_NAME => "MUNHAUSEN",
SOS_ID => "555554444", YEARLY_SAL => 25_000.00,
Y_T_DATE_TOTAL => 13_000.00);
```

Изменение значения дискриминанта в объекте комбинированного типа (т.е. структуре) разрешается, если всем другим компонентам этой структуры присваиваются новые значения, например:

```
CURR_EMPLOYEE := ( LAB_KIND      => HOURLY,
F_NAME          => "HARDY12345",
L_NAME          => "LAURENTINI54321",
SOS_ID          => "123456789",
HOURLY_PAY      => 9.75,
HOURS_WORKED    => 45.5,
OVERTIME_AMT    => 4.0 );
```

Следующая программа иллюстрирует применение дискриминантов и комбинированных типов с вариантами. Программа считывает данные об оценках студентов, полученных в конце семестра. Допустимы такие оценки<sup>1)</sup>: A, A-, B+, B, B-, C+, C, C-, D+, D, F, NC (no credit – нет оценки), W (withdrawal – студент прекратил заниматься по этому курсу), I (incomplete – не прошел достаточное количество курсов) и P (pass – зачет).

Для каждой категории студентов программа должна отображать среднеарифметическую оценку в баллах в соответствии со следующей шкалой<sup>2)</sup>: A – это 4.0, A- – это 3.7, B+ – это 3.3, ..., D – это 1.0, F – это 0. Должно соблюдаться следующее правило: для курсов, относящихся к основной специализации студента, требуется оценка не хуже C-, в противном случае необходимо повторное обучение по этому курсу, о чем программа должна выдать соответствующее сообщение. Для основных курсов или курсов, занимающих второе место по важности, оценка P не разрешается. Студент, который не должен получать степень бакалавра (nondegree student), т.е. занимающийся на курсах повышения квалификации, может получить любую оценку. Вольнослушателям (auditing students) оценки не выставляются. Студент, обучающийся с целью получения диплома бакалавра (degree student), должен иметь среднеарифметическую оценку не ниже 1.5, иначе ему дается некоторый испытательный срок. Для студентов, не получающих диплома бакалавра, следует также выдать суммарную оценку, а для вольнослушателей – общее количество часов посещенных занятий. Курсы высокого уровня, номера которых больше 100, вольнослушателям посещать не разрешается.

<sup>1)</sup> Это оценки, принятые за рубежом. – Прим. перев.

<sup>2)</sup> Полные сведения о соответствии буквенных и цифровых оценок приведены в тексте программы GR\_POINT\_AVE. – Прим. перев.

Исходная информация по каждому студенту дается в двух строках. В первой из них приводятся такие данные: количество учебных курсов, которые проходит данный студент (поз. 1–2), вид обучения студента – DEGREE, NONDEGREE или AUDITOR (поз. 3–15), личный номер студента (поз. 16–24), фамилия студента (поз. 25–39). Если студент обучается для получения степени бакалавра, то в поз. 40–43 содержится обозначение главного предмета, по которому специализируется студент, а в поз. 44–47 – обозначение второго по важности предмета. Во второй строке приводятся данные, касающиеся учебных курсов. Для вольнослушателей там задаются сведения об обозначении курса, кодовый номер курса, количество учебных часов. Для остальных студентов там приводятся данные об обозначении курса, его кодовый номер, количество учебных часов и оценка. Эта информация занимает 10 позиций для вольнослушателей и 12 позиций для остальных студентов. Конец потока данных отмечается записью, которая в поле количества учебных курсов содержит число 99.

### Программа GR\_POINT\_AVE

```
with TEXT_IO; use TEXT_IO;
procedure GR_POINT_AVE is
type COURSE_INFO is
  record
    CR_ID   : STRING(1 .. 4);
    -- Пример: "ENGL".
    CR_NO   : STRING(1 .. 4);
    -- Пример: "117 "
    CR_CRDT : NATURAL;
    -- Пример: 3.
  end record;
type COURSE_LIST is array (NATURAL range <>)
  of COURSE_INFO;
type COURSE_N_GRADE_INFO is
  record
    DESCR   : COURSE_INFO;
    CR_GRADE : STRING(1 .. 2);
    -- Пример: "A-".
  end record;
type TRANSCRIPT is array (NATURAL range <>)
  of COURSE_N_GRADE_INFO;
type STUD_KIND is (DEGREE, NON_DEGREE, AUDITOR);
package ST_TYPE_IO is new
  ENUMERATION_IO(STUD_KIND);
use ST_TYPE_IO;
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
type STUDENT (ST_TYPE : STUD_KIND := DEGREE ;
               NO_COURSES : NATURAL ) is
  record
    ST_ID   : STRING (1 .. 9);
    ST_NAME : STRING (1 .. 15);
    case ST_TYPE is
      when DEGREE =>
        ST_MAJOR : STRING(1 .. 4);
        ST_MINOR  : STRING(1 .. 4);
        ST_COURSES : TRANSCRIPT(1 .. NO_COURSES);
      when NON_DEGREE =>
        ST_COURSES : TRANSCRIPT(1 .. NO_COURSES);
      when AUDITOR =>
        CRS_AUDITED : COURSE_LIST(1 .. NO_COURSES);
```

```

end case;
end record;
CURR_STUD : STUDENT;
CURR_COURSE_INFO : COURSE_INFO := (" ", " ", " ",
                                     0);
CURR_NO_COURSES : NATURAL;
CURR_COURSES_N_GRADE : COURSE_N_GRADE_INFO :=
  ((" ", " ", " ", 0), " ");
CURR_ST_TYPE : STUD_KIND;
GRADE_POINT, GRADE_POINT_AVE : FLOAT;
package FLOAT_IO is new FLOAT_IO(FLOAT);
use FLOAT_IO;
VALID_GRADE, AVE_FLAG : BOOLEAN;
NO_IN_GRADE_POINT, NO_CREDITS : NATURAL;
SEC_CHAR : CHARACTER;
-- Значением этой переменной является второй
-- символ буквенной оценки (+, - или C).
procedure CHECK_GRADE is
begin
  -- Эта процедура проверяет правильность
  -- оценки, выставленной за учебный курс. Кроме
  -- того, если эту оценку можно использовать при
  -- вычислении средней оценки, то она по-
  -- мещается в переменную GRADE_POINT, а
  -- значение переменной AVE_FLAG устанавливается
  -- равным TRUE.
  VALID_GRADE := TRUE;
  SEC_CHAR := CURR_STUD.ST_COURSES(I).CR_GRADE(2);
  AVE_FLAG := TRUE;
  case CURR_STUD.ST_COURSES(I).CR_GRADE(1) is
    when "A" => case SEC_CHAR is
      when '+' => GRADE_POINT := 4.0;
      when '-' => GRADE_POINT := 3.7;
      when others => VALID_GRADE :=
        FALSE;
    end case;
    when "B" => case SEC_CHAR is
      when '+' => GRADE_POINT := 3.3;
      when ' ' => GRADE_POINT := 3.0;
      when '-' => GRADE_POINT := 2.7;
      when others => VALID_GRADE :=
        FALSE;
    end case;
    when "C" => case SEC_CHAR is
      when '+' => GRADE_POINT := 2.3;
      when ' ' => GRADE_POINT := 2.0;
      when '-' => GRADE_POINT := 1.7;
      when others => VALID_GRADE :=
        FALSE;
    end case;
    when "D" => case SEC_CHAR is
      when '+' => GRADE_POINT := 1.3;
      when ' ' => GRADE_POINT := 1.0;
      when others => VALID_GRADE :=
        FALSE;
    end case;
    when "F" => case SEC_CHAR is

```



```

        when ' ' => GRADE_POINT := 0.0;
        when others => VALID_GRADE :=
            FALSE;
    end case;
when "N" => case SEC_CHAR is
    when 'C' => AVE_FLAG := FALSE;
    when others => VALID_GRADE :=
        FALSE;
    end case;
when "W" | "I" | "P" =>
    case SEC_CHAR is
        when ' ' => AVE_FLAG := FALSE;
        when others => VALID_GRADE :=
            FALSE;
    end case;
when others => VALID_GRADE := FALSE;
end case;
end CHECK_GRADE;
begin
    GET (CURR_NO_COURSES, 2);
    while CURR_NO_COURSES /= 99
    loop
        GET (CURR_ST_TYPE, 13);
        -- Здесь реализуется один из трех возможных
        -- видов структуры CURR_STUD. Стали известны
        -- значения двух дискриминантов, и теперь
        -- структуру CURR_STUD можно проинициализи-
        -- ровать.
        case CURR_ST_TYPE is
            when DEGREE => CURR_STUD :=
                (ST_TYPE => CURR_ST_TYPE,
                 NO_COURSE => CURR_NO_COURSES,
                 ST_ID => "123456789",
                 ST_NAME => "123456789012345",
                 ST_MAJOR => "XXXX",
                 ST_MINOR => "YYYY",
                 ST_COURSES(1) ..
                 ST_COURSES(CURR_NO_COURSES) =>
                     CURR_COURSE_N_GRADE );
            when NON_DEGREE => CURR_STUD :=
                (ST_TYPE => CURR_ST_TYPE,
                 NO_COURSE => CURR_NO_COURSES,
                 ST_ID => "123456789",
                 ST_NAME => "123456789012345",
                 ST_COURSES(1) ..
                 ST_COURSES(CURR_NO_COURSES) =>
                     CURR_COURSE_N_GRADE );
            when AUDITOR => CURR_STUD :=
                (ST_TYPE => CURR_ST_TYPE,
                 NO_COURSE => CURR_NO_COURSES,
                 ST_ID => "123456789",
                 ST_NAME => "123456789012345",
                 CRS_AUDITED(1) ..
                 CRS_AUDITED(CURR_NO_COURSES) =>
                     CURR_COURSE_INFO );
        end case;
    end loop;
end;

```

```

GET ( CURR_STUD.ST_ID );
GET ( CURR_STUD.ST_NAME );
case CURR_ST_TYPE is
  when DEGREE =>
    GET(CURR_STUD.ST_MAJOR) ;
    GET(CURR_STUD.ST_MINOR) ;
    SKIP_LINE;
    for I in 1 .. CURR_NO_COURSES
      loop
        GET(CURR_STUD.ST_COURSES(I).DESC.CR_ID);
        GET(CURR_STUD.ST_COURSES(I).DESC.CR_NO);
        GET(CURR_STUD.ST_COURSES(I).DESC.
              CR_CRDT, 2);
        GET(CURR_STUD.ST_COURSES(I).CR_GRADE,2);
      end loop;
  when NON_DEGREE =>
    SKIP_LINE;
    for I in 1 .. CURR_NO_COURSES
      loop
        GET(CURR_STUD.ST_COURSES(I).DESC.CR_ID);
        GET(CURR_STUD.ST_COURSES(I).DESC.CR_NO);
        GET(CURR_STUD.ST_COURSES(I).DESC.
              CR_CRDT, 2);
        GET(CURR_STUD.ST_COURSES(I).CR_GRADE,2);
      end loop;
  when AUDITOR =>
    SKIP_LINE;
    for I in 1 .. CURR_NO_COURSES
      loop
        GET(CURR_STUD.CRS_AUDITED(I).CR_ID);
        GET(CURR_STUD.CRS_AUDITED(I).CR_NO);
        GET(CURR_STUD.CRS_AUDITED(I).CR_CRDT,2);
      end loop;
end case;
-- В данном месте программы информация о сту-
-- денте запомнена в переменной CURR_STUD.
-- Теперь проверим правильность оценок и
-- начнем вычисление средней оценки, если толь-
-- ко студент не является вольнослушателем.
case CURR_ST_TYPE is
  when DEGREE =>
    GRADE_POINT_AVE := 0.0;
    NO_IN_GRADE_POINT := 0;
    for I in 1 .. CURR_NO_COURSES
      loop
        CHECK_GRADE;
        -- Оценка за курс - это буква от
        -- F до A.
        if VALID_GRADE and AVE_FLAG
          then
            GRADE_POINT_AVE :=
              GRADE_POINT_AVE + GRADE_POINT;
            NO_IN_GRADE_POINT :=
              NO_IN_GRADE_POINT + 1;
            -- Это курс по основной дисциплине?
            -- Если да, то оценка должна быть
            -- выше, чем C-, т.е. GRADE_POINT

```

```

-- должна быть больше 1.7.
if CURR_STUD.ST_MAJOR =
  CURR_STUD.ST_COURCES(I).DESC.CR_ID
  and then
    GRADE_POINT < 1.7
  then
    PUT(" The course must be repeated");
    PUT(I);
    NEW_LINE;
  and if;
end if;
-- Вспомните, что для основного и вторич-
-- ного предмета не допускается оценка
-- P (сачет).
if (CURR_STUD.ST_MAJOR =
  CURR_STUD.ST_COURCES(I).DESC.CR_ID
  or
  CURR_STUD.ST_MINOR =
  CURR_STUD.ST_COURCES(I).DESC.CR_ID)
  and
  CURR_STUD.ST_COURCES(I).CR_GRADE
  = "P"
  then
    PUT(" No pass fail option");
    PUT(I);
  and if;
end loop;
if NO_IN_GRADE_POINT > 0
  then
    GRADE_POINT_AVE := GRADE_POINT_AVE /
      FLOAT(NO_IN_GRADE_POINT);
    PUT(GRADE_POINT_AVE, 5, 3);
    if GRADE_POINT_AVE < 1.7
      then
        PUT( " Probation ");
      and if;
    PUT(CURR_STUD.ST_NAME);
    NEW_LINE;
  and if;
when NON_DEGREE =>
  GRADE_POINT_AVE := 0.0;
  NO_IN_GRADE_POINT := 0;
  for I in 1 .. CURR_NO_COURSES
    loop;
    CHECK_GRADE;
    if VALID_GRADE and AVE_FLAG
      then
        GRADE_POINT_AVE :=
          GRADE_POINT_AVE + GRADE_POINT;
        NO_IN_GRADE_POINT :=
          NO_IN_GRADE_POINT + 1;
      and if;
    end loop;
  if NO_IN_GRADE_POINT > 0
    then
      GRADE_POINT_AVE := GRADE_POINT_AVE /
        FLOAT(NO_IN_GRADE_POINT);
    end if;
  end if;
end if;

```

```

    PUT(GRADE_POINT_AVE, 5, 3);
  end if;
  PUT(CURR_STUD.ST_NAME);
  NEW_LINE;
  when AUDITOR =>
    NO_CREDITS := 0;
    for I in 1 .. CURR_NO_COURSES
      loop
        if CURR_STUD.CRS_AUDITED(I).CR_NO
          > 99
        then
          PUT( " Course do not allowed ");
        else
          NO_CREDITS := NO_CREDITS +
            CURR_STUD.CRS_AUDITED(I).CR_CRDT;
        end if;
      end loop;
    PUT(NO_CREDITS, 3);
    PUT(CURR_STUD.ST_NAME);
    NEW_LINE;
  end case;
  GET (CURR_NO_COURSES, 2);
end loop;
end GR_POINT_AVE;

```

## УПРАЖНЕНИЯ

1. Модифицируйте программу NAME\_PHONE следующим образом. Первые символы первого имени и фамилии нужно печатать прописными буквами, а остальные символы — строчными.

2. Пусть на вход подаются целые действительные литералы (по одному литералу в строке). Используйте процедуры из программы NAME\_PHONE для проверки корректности каждого литерала. Программа должна выводить числовой литерал и сообщение об его правильности или неправильности. Сведения о литералах были представлены в гл. 1.

3. Напишите программу, которая считывает даты и затем печатает их в стандартном формате. Каждая дата располагается в одной входной строке и может иметь несколько форматов. В начале строки идет число или месяц, затем соответственно — месяц или число, а в конце строки — год. Перед каждым из трех значений стоит по меньшей мере один пробел. Перед годом может стоять запятая и по крайней мере один пробел. Названия месяцев<sup>1)</sup> можно сокращать, но не более чем до трех символов. Если они сокращены, то после последнего символа названия месяца должна стоять точка. Для названий месяцев разрешается употреблять и прописные, и строчные буквы. После цифр, обозначающих число, можно по желанию ставить суффикс th<sup>2)</sup>, а если числа — 1, 2, 3, то цифра и суффикс принимают вид: 1st, 2nd, 3rd. В суффиксах можно использовать буквы обоих регистров. Если число стоит перед названием месяца, то между днем и месяцем ставится слово of. Если же название месяца размещается первым, то между месяцем и числом может стоять артикль the. Дни имеют номера от 1 до 31, а в обозначении года должно быть ровно четыре цифры. Вот примеры допустимых дат:

```

15th of November, 1975
Nov. 17, 1984
Sept. 12 1985
1st JULY 1985
DEcemb. the 2nd 1987

```

<sup>1)</sup> Естественно, английские названия. — *Прим. перев.*

<sup>2)</sup> Обозначающий порядковые числительные в английском языке. — *Прим. перев.*

А вот примеры некорректных дат:

3 Jun 1985	После Jun нет точки
4th of July 87	В обозначении года – только две цифры
Sept. the 4th, 1988	Перед обозначением года нет пробела

Для каждой введенной строки следует напечатать дату и признак ее корректности (valid или invalid). Признаком конца данных служит строка с точкой в первой позиции. При написании этой программы можно воспользоваться процедурами из программы NAME\_PHONE.

4. Напишите программу, подсчитывающую пенсионные взносы служащих. Взносы платят только служащие, относящиеся к категориям HOURLY (почасовая оплата) и SALARIED (оклад), а служащие, принадлежащие к категории SUBCONTRACTOR (работа по договору), взносов не платят. Входные данные для типов LABOR и EMPLOYEE, определенных в разд. 4.3, имеют формат:

Позиции	Данные
1–15	LAB_KIND (вид служащего)
16–25	F_NAME (имя)
26–40	L_NAME (фамилия)
41–49	SOS_ID (номер по соцстраху)

Далее в зависимости от категории служащего, следуют два действительных числа, три действительных числа или целое и действительное число (см. разд. 4.3). Сведения о каждом служащем занимают одну строку, а в первой строке приведено число, равное количеству строк с данными о служащих. Размер пенсионного взноса составляет 10% от значения переменной Y\_T\_DATE\_TOTAL (доход) для служащих категории SALARIED (оклад) и 8% от номинальной заработной платы для служащих категории HOURLY (почасовая оплата). Для служащих с почасовой оплатой сумма номинального дохода вычисляется по формуле

$$\text{HOURLY\_PAY} * (\text{HOURS\_WORKED} + 1.5 * \text{OVERTIME\_HRS})$$

Здесь приняты обозначения:

HOURLY\_PAY – оплата за час работы;  
HOURS\_WORKED – отработано часов;  
OVERTIME\_HRS – отработано сверхурочных часов.

Размер взноса ограничен суммой в 5000 долл. Служащий может одновременно фигурировать и как SALARIED, и как HOURLY, так как должность его в компании может неоднократно изменяться. Входные данные следуют в произвольном порядке. Программа должна выводить имя каждого служащего и размер его пенсионного взноса.

5. Студенты часто не соглашались со своей итоговой оценкой и просят объяснить, как она вычисляется. Для проверки правильности итоговой оценки измените программу GR\_POINT\_AVE следующим образом. Сведения о студентах, обучающихся на степень бакалавра, и о студентах, не получающих этой степени, разделяются на три порции. Первые две порции идентичны входным данным исходной версии программы GR\_POINT\_AVE. В третьей части данных приводится информация о каждой контрольной работе, использовавшейся при вычислении итоговой оценки. Для каждого учебного курса, который был изучен студентом и по которому выставлена буквенная оценка (от A до F), задается входная строка следующего формата:

Позиции	Данные
1–4	Обозначение предмета
5–8	Кодовый номер учебного курса
9–10	Количество контрольных работ
11–12	Вес оценки за первую контрольную работу (например, 15 обозначает 15% от итоговой оценки)
13–14	Буквенная оценка первой контрольной работы (A, A–, ..., F)
15–18, 19–22 и т.д.	Информация о второй, третьей и т.д. контрольных работах

Программа GR\_POINT\_AVE теперь должна выполнять дополнительные действия. Каждую итоговую оценку следует сравнивать со средневзвешенной оценкой, вычисленной на основании строк с информацией о контрольных работах. Необходимо напечатать сообщение, свидетельствующее о правильности итоговой оценки. Например, для сообщения Final grade B (итоговая оценка – B) строка с данными о контрольных работах имеет вид

30A 20B 20B 30C

а средневзвешенная оценка вычисляется так:

$$(30 * 4.0 + 20 * 3.0 + 20 * 3.0 + 30 * 2.0) / 4 = 3.0 \text{ (соответствует B)}$$

В этом примере итоговая оценка правильная. Если средневзвешенная оценка попадает в интервал между двумя буквенными оценками, то берется более высокая буквенная оценка.

## Подпрограммы: процедуры и функции

### 5.1. ПРОЦЕДУРЫ

Как отмечалось в гл. 1, программы на языке Ада состоят из одного или более *программных сегментов*. В данной главе будет рассмотрен один из видов программных сегментов — *подпрограммы*. При написании программ, представленных в первых четырех главах книги, уже применялась одна из разновидностей подпрограмм, называемая *процедурой*. Существует и вторая разновидность подпрограмм, которая называется *функцией*. Функции будут представлены в следующем разделе.

Использовавшиеся ранее процедуры имели один и тот же общий вид. Все они определяли *тело подпрограммы*:

```
procedure идентификатор_процедуры is
  возможные_объявления
begin
  последовательность_операторов
end идентификатор_процедуры;
```

Вот простейший пример:

```
procedure NOTHING is
begin
  NULL;
end NOTHING;
```

В этой процедуре отсутствует *формальная часть*, в которой приводится список *спецификаций параметров*, заключенный в скобки. Если же формальная часть присутствует, то она должна располагаться сразу за идентификатором процедуры. Спецификации параметров отделяются друг от друга символом «;». Указать спецификацию параметров несложно. Для этого надо задать список переменных, за которым ставится двоеточие и указывается их тип или подтип.

**Пример.** Следующая процедура демонстрирует запись тела подпрограммы с формальной частью

```
procedure SOMETHING ( I,J : INTEGER ;
                      K : STRING ) is
LOC : INTEGER := 4;
begin
LOC := I * J * LOC ;
if LOC > 100 then
  PUT ( " Product exceeds limit " ) ;
end if;
if K ( 1 .. 5 ) = K ( 6 .. 10 ) then
  PUT ( "Repeated string " );
end if;
end SOMETHING;
```

В процедуре SOMETHING имеется формальная часть (I, J : INTEGER; K : STRING). Здесь есть две спецификации параметров. Первая из них — это спецификация I, J : INTEGER, а вторая, располагающаяся после символа «;», — спецификация K : STRING. В первой спецификации параметров список переменных I, J состоит из двух переменных, разделенных запятой. Их тип или подтип — INTEGER. Во второй спецификации параметров список состоит всего лишь из одной переменной K и обозначения типа STRING. Это обозначение является одним из способов, которыми указываются типы или подтипы (см. гл. 2).

### 5.1.1. Формальные параметры

Переменные, которые указываются в этих списках, называются *формальными параметрами*. Так, I, J и K — это формальные параметры. Они известны только в пределах тела процедуры. Тип формального параметра задается обозначением типа после двоеточия в спецификации параметров.

Выполнение процедуры заканчивается при достижении зарезервированного слова end, за которым следует идентификатор процедуры. Выполнение ранее приведенных процедур заканчивалось именно таким способом. Однако выполнение процедуры может быть закончено в любом месте ее последовательности операторов с помощью оператора возврата

return;

Например, можно изменить процедуру SOMETHING, указав другую возможную точку выхода из нее:

```
procedure SOMETHING2 ( I, J : INTEGER ;
                      K : STRING ) is
  LOC : INTEGER := 4;
begin
  if I > 10 then
    return;
  end if;
  LOC := I * J * LOC ;
  if LOC > 100 then
    PUT ( " Product exceeds limit " );
  end if;
  if K ( 1 .. 5 ) = K ( 6 .. 10 ) then
    PUT ( " Repeated string " );
  end if;
end SOMETHING2;
```

Более полное определение спецификации параметров включает указание вида связи формальных параметров. Возможны следующие *виды связи*: in (входной), in out (изменяемый), out (выходной). Если вид связи не указывается, то по умолчанию принимается значение in. Таким образом, в процедуре SOMETHING все формальные параметры (I, J и K) входные, т.е. имеют вид связи in.

В языке Ада не разрешается изменение значений формальных параметров с видом связи in. Поэтому оператор

```
I := I * J;
```

в теле процедуры SOMETHING2 будет считаться ошибочным, поскольку значение I в этом случае изменится.

По желанию программиста формальным параметрам можно присвоить начальные значения. Для этого за обозначением типа ставятся символ присваивания := и



выражение. Такая инициализация разрешена только для формальных параметров с видом связи in.

Проиллюстрируем введенные понятия на примере новой версии нашей процедуры—SOMETHING3:

```

procedure SOMETHING3 ( I,J : in out INTEGER ;
                      K : in STRING := "CREATURES " ) is
  LOC : INTEGER := 4;
begin
  if I > 10 then
    return;
  end if;
  LOC := I * J * LOC ;
  if LOC > 100 then
    PUT ( " Product exceeds limit " ) ;
  end if;
  if K ( 1 .. 5 ) = K ( 6 .. 10 ) then
    PUT ( " Repeated string " );
  end if;
end SOMETHING3;

```

В процедуре SOMETHING3 переменные I и J имеют вид связи in out. Переменная K имеет вид связи in, у нее есть начальное значение, которое она будет принимать по умолчанию.

В процедуре объявлена еще одна переменная—LOC. Она, так же как и формальные параметры, известна только внутри процедуры. Но в отличие от них её нельзя поставить в соответствие каким-либо внешним переменным.

## 5.1.2. Вызов процедур

Процедуру можно вызвать несколькими способами. В представленных к настоящему моменту программах тела вызываемых процедур размещались внутри программ, обращающихся к этим процедурам. Такой способ применим, если тело процедуры размещается в вызывающей программе до момента обращения к этой процедуре.

Если же процедура вызывается из программы на Аде, а тело этой процедуры располагается после места ее вызова, то в декларативной части вызывающей программы следует задать объявление вызываемой процедуры. Объявление подпрограммы заключается только в указании ее спецификации, за которой следует символ «;». Спецификация процедуры—это зарезервированное слово *procedure*, за которым следуют идентификатор процедуры и формальная часть (если она имеется).

Вне зависимости от того, нуждается ли процедура в объявлении, ее можно вызвать, если указать имя этой процедуры в вызывающей программе. Если вызываемая процедура имеет формальную часть, то при вызове за именем процедуры должен располагаться список фактических параметров, заключенный в скобки.

В процессе выполнения вызываемой процедуры происходит согласование формальных и фактических параметров, обработка объявлений в процедуре, а затем выполнение ее последовательности операторов с использованием значений фактических параметров. Выполнение процедуры заканчивается по операторам *end* или *return*, если только в процессе выполнения не возникли исключительные ситуации. В предыдущих примерах тела процедур располагались так, что для обращения к ним не требовались предварительные объявления этих процедур.

Следующие примеры программ демонстрируют вызов процедур с формальной частью и с использованием объявлений процедур. Кроме того, определяется формальный параметр с видом связи *out*.

## Процедура ENVELOP

```

with TEXT_IO; use TEXT_IO;
procedure ENVELOP is
  II, JJ : INTEGER;
  KK : STRING(1 .. 10);
  LL : STRING(1 .. 10);
  procedure SOMETHING4(I, J : in out INTEGER;
    K : in STRING := "CREATURES ";
    L : out STRING ) is
    LOC : INTEGER := 4;
  begin
    if I > 10 then
      return;
    end if;
    I := I * J * LOC; LOC := I;
    if LOC > 100 then
      PUT( " Product exceeds limit ");
    end if;
    if K(1 .. 5) = K(6 .. 10) then
      PUT( " Repeated string "); L := K;
    else
      L(1..10) := K(1 .. 5) & K(1 .. 5);
    end if;
  end SOMETHING4;
  procedure INVOKES is
  begin
    SOMETHING4(II, JJ, KK, LL);
    --Здесь вызывается процедура SOMETHING4. Факти-
    --ческими параметрами являются : II, JJ, KK и LL.
  end INVOKES;
begin
  II := 5 JJ := 8;
  KK := "XXXXXXXXXXXX";
  INVOKES;
end ENVELOP;

```

А вот версия той же самой процедуры ENVELOP (Объемлощая) с объявлением процедуры SOMETHING4:

Процедура ENVELOP, в которой используется объявление процедуры

```

with TEXT_IO; use TEXT_IO;
procedure ENVELOP is
  II, JJ : INTEGER;
  KK : STRING(1 .. 10);
  LL : STRING(1 .. 10);
  procedure SOMETHING4(I, J : in out INTEGER;
    K : in STRING := "CREATURES ";
    L : out STRING);
  -- Здесь помещено об'явление процедуры SOMETHING4,
  -- оно требуется, поскольку процедура INVOKES
  -- вызовет процедуру SOMETHING4 до того, как
  -- будет определено тело процедуры SOMETHING4.
  procedure INVOKES is
  begin
    SOMETHING4(II, JJ, KK, LL);
  end INVOKES;

```

```

procedure SOMETHING4(I, J : in out INTEGER;
                    K : in STRING := "CREATURES ";
                    L : out STRING) is
    LOC : INTEGER := 4;
begin
    if I > 10 then
        return;
    end if;
    LOC := I * J * LOC; I := LOC;
    if LOC > 100 then
        PUT(" Product exceeds limit");
    end if;
    if K(1 .. 5) = K(6 .. 10) then
        PUT(" Repeated string"); L:=K;
    else
        L(1..10) := K(1..5) & K(1..5);
    end if;
end SOMETHING4;
begin
    II :=5; JJ :=8;
    KK := "XXXXXXXXXX";
    INVOKES;
end ENVELOP;

```

### 5.1.3. Согласование формальных и фактических параметров

Как уже отмечалось, при вызове процедуры производится связывание фактических и формальных параметров. При обращении к процедуре SOMETHING4 (Нечто\_4) из процедуры INVOKES (Вызывающая) фактические параметры II, JJ, KK связываются с формальными параметрами I, J, K. Типы формальных и фактических параметров должны в точности совпадать. В данном случае обязательна идентичность типов для II и I, JJ и J, KK и K.

Вид связи формальных параметров определяет порядок действий со значениями связанных друг с другом формальных и фактических параметров. В нашем примере для формального параметра K, имеющего вид связи in, значение фактического параметра KK передается в подпрограмму. (Здесь в подпрограмму SOMETHING4 передается значение XXXXXYYYYY.) В подпрограмме это значение можно использовать, но его нельзя изменять. После выполнения процедуры обратно в вызывающую программу не передается никакого нового значения для фактического параметра. Иными словами, значение фактического параметра выполняет роль константы в вызываемой подпрограмме.

Если вид связи формального параметра – out, то при вызове процедуры формальному параметру не передается значение соответствующего фактического параметра. Но после окончания выполнения процедуры получившееся значение формального параметра передается соответствующему фактическому параметру. В процедуре SOMETHING4 формальный параметр L имеет вид связи out, и после завершения процедуры его значение передается переменной LL. В данном случае LL получит значение XXXXXXXXXXXX.

Если вид связи формального параметра – in out, то вызываемая процедура начинает выполняться со значением формального параметра, которое будет равно значению соответствующего фактического параметра. В процедуре ENVELOP фактические параметры II и JJ имеют значения 5 и 8. После обращения к процедуре SOMETHING4 она начнет выполняться с начальными значениями I и J, равными 5 и 8.

Значения формальных параметров с видом связи in out можно изменять в вызываемой процедуре. Когда процедура закончится, их новые значения будут переданы соответствующим фактическим параметрам. Во время выполнения процедуры SOMETHING4 ее формальный параметр I получает новое значение 160. После завершения процедуры фактический параметр II получит новое значение 160. Значение формального параметра J, которое не меняется в процессе работы процедуры и остается равным 8, будет передано фактическому параметру JJ. Присваивание новых значений формальным параметрам с видом связи in out разрешено.

Разумеется, тип формальных параметров может быть любым из знакомых читателю — регулярным, комбинированным, ссылочным и т. д. Формальные параметры могут также принадлежать и к приватным типам (см. гл. 7).

Если работа программы закончится аварийно, скажем если будет возбуждена исключительная ситуация, то результирующее значение фактического параметра будет непредсказуемо.

Фактическим параметром могут служить переменная или значение выражения. Если же соответствующий формальный параметр имеет вид связи in out или out, то фактическим параметром может являться только имя переменной или выражение, выполняющее преобразование типа для переменной. Например, можно вызвать процедуру SOMETHING4 так:

```
SOMETHING4 (II, JJ, "ABCDEABCDE", LL);
```

поскольку формальный параметр K — входной. Однако обращение вида:

```
SOMETHING4 (4, JJ, "ABCDEABCDE", LL);
```

будет недопустимо, так как формальный параметр I имеет вид связи in out, а соответствующий ему фактический параметр обязан быть именем переменной или преобразованием типа для нее.

### 5.1.4. Поименованные параметры при обращениях к процедурам

При вызове процедуры SOMETHING4 для передачи значений фактических параметров использовалась позиционная форма записи. В этом случае каждый формальный параметр связывается с тем фактическим параметром, который расположен в той же самой позиции в списке фактических параметров, что и формальный.

В языке Ада программист может явно задавать имена формальных параметров, соответствующих фактическим, при вызове процедуры. При этом программист должен указать имя формального параметра, за которым следует составной символ «= >», и затем фактический параметр. Ниже даны эквивалентные варианты обращения к процедуре SOMETHING4:

```
SOMETHING4 (II, JJ, KK, LL);
SOMETHING4 (I => II, J => JJ, K => KK, L => LL);
SOMETHING4 (J => JJ, K => KK, I => II, L => LL);
```

В двух последних вызовах используется *связывание поименованных параметров*. При связывании поименованных параметров порядок их следования в списке не имеет значения.

В одном и том же обращении можно сочетать поименованные и позиционные параметры. В этом случае сначала следует располагать позиционные параметры. После того как указан один поименованный параметр, все остальные параметры обязаны быть также поименованными. Пример правильного вызова:

```
SOMETHING4 (II, JJ, L => LL, K => KK);
```

Однако вызов

```
SOMETHING4 (II, L => LL, K => KK, JJ);
```

неверен, поскольку за поименованным параметром следует позиционный.

Если формальный параметр имеет начальное значение по умолчанию, то задавать этот параметр при вызове процедуры необязательно. Например, будет корректным такое обращение к процедуре:

```
SOMETHING4 (I => II, L => LL, J => JJ);
```

Для формального параметра K отсутствует соответствующий фактический параметр, поэтому будет использовано принимаемое по умолчанию начальное значение CREATURES (Существа).

### 5.1.5. Уточнения для формальных параметров

Уточнения, задаваемые для формальных параметров, должны соблюдаться и фактическими параметрами согласно следующим правилам. Если вид связи формального параметра — in или in out и этот параметр принадлежит к скалярному типу, то значения фактических параметров в момент, непосредственно предшествующий вызову процедуры, должны удовлетворять всем уточнениям диапазонов значений формальных параметров. Если вид связи формального параметра, опять-таки принадлежащего к скалярному типу, — out или in out, то значение формального параметра<sup>1)</sup> должно удовлетворять всем уточнениям для фактических параметров.

Если параметр относится к регулярному, комбинированному или приватному с дискриминантами (см. гл. 7) типу, то независимо от вида связи все уточнения для формального параметра должны быть соблюдены фактическими параметрами. Это относится к моменту времени, непосредственно предшествующему вызову процедуры. Применение некоторых из правил для уточнений иллюстрируется процедурой MAIN\_CALL (Главн\_вызов).

#### Процедура MAIN\_CALL

```
procedure MAIN_CALL is
  type REAL_TABLE is array(POSITIVE range <> )
                        of FLOAT;
  SOME_TABLE : REAL_TABLE(1 .. 12)
                := (1 .. 12 => 5.0);
                -- Здесь имеется уточнение диапазона
                -- индексов.
  CURR_TABLE : REAL_TABLE(1 .. 10)
                := (1 .. 10 => 5.0);
                -- Это - еще одно уточнение диапазона
                -- индексов.
  type REC_DISCR(STR_LENGTH : range 1 .. 255 := 80)
    is
      record
        LINE_CONT : STRING(1 .. STR_LENGTH);
      end record;
  LONG_LINE : REC_DISCR (90);
  -- Здесь имеется уточнение дискриминанта.
  REG_LINE : REC_DISCR (75);
  -- Это - еще одно уточнение дискриминанта.
  type DIGIT_COUNT is range 0 .. 9;
  subtype FEW_CHOICES is DIGIT_COUNT range 1 .. 5;
```

<sup>1)</sup> В момент нормального окончания выполнения процедуры. — Прим. перев.

```

II : DIGIT_COUNT;
JJ : FEW_CHOICES;
procedure CONSTR_PARAM(
  I : DIGIT_COUNT;
  J : in out DIGIT_COUNT;
  X : in out REAL_TABLE;
  FOR_LINE : in out REC_DISCR) is
begin
  J := I + 2;
  X := X + X;
  FOR_LINE(20 .. 30) := FOR_LINE(40 .. 50);
and CONSTR_PARAM;
begin
  REG_LINE(1 .. 10) := "0123456789";
  for L in 11 .. 75
  loop
    REG_LINE(L) := ' ';
  end loop;
  LONG_LINE := (1 .. 90 => 'A');
  II := 5; JJ := 3;
  CONSTR_PARAM(2, JJ, CURR_TABLE, REG_LINE);
  -- Это - правильный вызов, все уточнения соблюдаются.
  CONSTR_PARAM(13, JJ, CURR_TABLE, REG_LINE);
  -- Здесь нарушено уточнение диапазона значений.
  -- Значение фактического параметра, 13, превышает
  -- верхнюю границу диапазона допустимых значений
  -- формального параметра I, равную 9.
  -- Возникает исключительная ситуация
  -- CONSTRAINT_ERROR.
  CONSTR_PARAM(II, JJ, CURR_TABLE, REG_LINE);
  -- Здесь нарушается уточнение диапазона значений.
  -- В результате вычислений формальный параметр J
  -- получает значение 7. Это значение передается
  -- обратно фактическому параметру JJ, которое не
  -- может быть больше чем 5. Возникает исключитель-
  -- ная ситуация CONSTRAINT_ERROR.
  CONSTR_PARAM(3, JJ, CURR_TABLE, LONG_LINE);
  -- Здесь уточнение дискриминанта формального
  -- параметра не соблюдается фактическим пара-
  -- метром. Возбуждается исключительная ситуа-
  -- ция CONSTRAINT_ERROR.
  CONSTR_PARAM(3, JJ, SOME_TABLE, REG_LINE);
  -- Здесь уточнение диа-
  -- пазона индексов формального параметра не
  -- соблюдается фактическим параметром, и по-
  -- этому возникает исключительная ситуация
  -- CONSTRAINT_ERROR.
end MAIN_CALL;

```

## 5.2. ФУНКЦИИ

Теперь обратимся ко второй разновидности подпрограмм Ады — функциям. *Функция* в отличие от процедуры должна вырабатывать некоторое значение. Кроме того, вызовы функций представляют собой простейшие выражения, в то время как вызовы процедур являются самостоятельными операторами. Тело функции имеет следующий вид:

function      идентификатор\_или\_обозначение\_операции  
                   необязательная\_формальная\_часть  
                   return тип\_или\_подтип is

возможные\_объявления

begin

последовательность\_операторов

— В последовательности операторов обязательно

— должен быть оператор возврата return.

end идентификатор\_или\_обозначение\_операции;

Как и для случая процедур, текст, предшествующий зарезервированному слову is, образует спецификацию подпрограммы (здесь — функции). Если для названия функции используется обозначение операции, то оно должно представлять собой строку символов. Если присутствует формальная часть, то ее формат должен быть таким же, как и у формальной части процедуры, за исключением того, что единственно допустимым видом связи для формальных параметров функции может быть in (входной).

В теле функции обязательно должен присутствовать оператор возврата return. Функция заканчивается только после выполнения оператора return. Для функций используется иная форма оператора возврата, чем для процедур. Здесь наличие выражения, стоящего после зарезервированного слова return, обязательно. Эта форма такова:

return выражение;

Значение выражения в операторе возврата — это значение, которое вырабатывает функция.

**Пример** тела функции и ее вызова:

```
with TEXT_IO; use TEXT_IO;
procedure CALL_OF_FUNCTION is
  II : INTEGER;
  JJ : FLOAT := 5.0;
  function DOUBLE ( I : INTEGER := -1;
                    J : FLOAT   := 1.0 )
    return FLOAT is
  K : INTEGER := 5;
  begin
    if I > 0
    then
      return FLOAT( 2 * K * I);
    else
      return 2.0 * J;
    end if;
  end DOUBLE;
  begin
    for M in 1 .. 10
    loop
      II := (-1) ** M;
      JJ := DOUBLE(II, JJ);
      if II > JJ then
        PUT(" First argument exceeds the second ");
      end if;
    end loop;
  end CALL_OF_FUNCTION;
```

Здесь идентификатором функции является слово DOUBLE (Двойной). Выход из функции DOUBLE осуществляется по одному из двух операторов возврата. Функция

вырабатывает значение типа `FLOAT`. Имеется формальная часть, в которой приведен список из двух параметров: `I` типа `INTEGER` и `J` типа `FLOAT`. Их вид связи — `in`, поскольку никакие другие виды связи параметров для функций не разрешены. Эти параметры имеют начальные значения, которые они будут принимать по умолчанию.

Поскольку оба формальных параметра из предыдущего примера имеют начальные умалчиваемые значения, то будут корректными следующие операторы присваивания:

<code>JJ := DOUBLE;</code>	Получается значение 2.0
<code>JJ := DOUBLE</code> <code>(J =&gt; JJ);</code>	По умолчанию <code>I := -1</code> , и в результате выполнения получится удвоенное значение <code>JJ</code>
<code>JJ := DOUBLE</code> <code>(I =&gt; II);</code>	Будет использоваться умалчиваемое значение <code>J := 1.0</code> . В результате вычислений получится величина, равная десятикратному значению <code>II</code>

В разд. 4.2 мы упоминали, не вдаваясь в детали, что функции могут вырабатывать значения в виде массивов. В этом случае функции можно использовать как имена индексированных компонент. Следующий пример иллюстрирует это положение.

**Пример.** Эта программа демонстрирует использование функции в качестве имени индексированной компоненты.

```
with TEXT_IO; use TEXT_IO;
procedure GET_MULTIPLE_TABLES is
  type TABLE is array(POSITIVE range <>)
    of INTEGER;
  type TWO_DIM_TABLE is array (POSITIVE range <> ,
    POSITIVE range <> )
    of INTEGER;

  II, JJ : INTEGER;
  CURR_TWO_TABLE : TWO_DIM_TABLE ( II, II );
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  function FILL_TABLE(I, J : INTEGER)
    return TABLE is
    FUN_TABLE : TABLE(I);
  -- Эта функция помещает чередующиеся значения
  -- J и -J в таблицу FUN_TABLE, начиная с J,
  -- если J — нечетное.
  begin
    for L in 1 .. I
      loop
        FUN_TABLE(L) := (-1) ** (I + J) * J;
      end loop;
    return FUN_TABLE;
  end FILL_TABLE;
begin
  GET(II); GET(JJ);
  for LL in 1 .. II
    loop
      -- Если JJ положительно, то поместить
      -- таблицу, полученную с помощью функции
      -- FILL_TABLE, в главную диагональ дву-
      -- мерного массива CURR_TWO_TABLE.
      if JJ > 0
        then
          CURR_TWO_TABLE(LL, LL) :=
            FILL_TABLE(II, JJ) (LL);
          -- Это индексированная компонента.
          -- Вызов функции FILL_TABLE(II, JJ)
```



```

-- дает массив, а LL - это значение
-- нужной компоненты массива.
-- Обратите внимание на то, что
-- функция вычисляется каждый раз
-- заново при повторном выполнении
-- оператора присваивания. (Это -
-- не слишком эффективно для данной
-- задачи.)
end if;
end loop;
end GET_MULTIPLE_TABLE;

```

Когда в разд. 4.2 вводилась общая форма для селектируемых компонент, то отмечалось, что имя функции может представлять собой имя, вырабатывающее структуру. В этом случае селектируемая компонента может иметь вид `имя_функции.идентификатор_компоненты`

Эта ситуация демонстрируется в следующем разделе на примере функции `FIND_COUP_DATE` (Найти дату купона).

### 5.3. ПРИКЛАДНАЯ ПРОГРАММА, В КОТОРОЙ ПРИМЕНЯЮТСЯ ФУНКЦИИ И ПРОЦЕДУРЫ

В данном разделе новые введенные понятия будут проиллюстрированы на примере программы `ACCR_INTEREST` (Наросшие проценты). Эта программа вычисляет выросшие проценты по ценным бумагам правительства США, имеющим купон. Наличие длинных или коротких купонов не учитывается. Формула для вычисления выросших процентов такова:

$$\text{Accrnt} := \text{Rate} / 2 * \text{nofdays} / \text{totaldays} * \text{parvalue}$$

Здесь приняты следующие обозначения:

`Accrnt` - выросшие проценты;  
`nofdays` - количество дней, прошедших с момента последней выплаты процентов по купону;  
`Rate` - годовой процент (десятичное число, например 12.5);  
`totaldays` - количество дней между датой последней выплаты и датой следующей за ней выплаты процентов по купону;  
`parvalue` - номинальное достоинство ценной бумаги (будем считать, что это 100 долл.).

Выплата по купонам производится два раза в год. На вход программы поступают строки, несущие следующую информацию:

Позиции	Наименование	Описание
1-7	<code>Rate</code> (годовой процент)	Представлен десятичным числом
8-13	<code>Mat_date</code> (срок выкупа)	Срок выкупа ценных бумаг в формате ГГММДД. Например, 950515 означает 15 мая 1995 г. Обратите, что для этого конкретного срока выкупа платежи по купонам запланированы на 15 мая и 15 ноября каждого года
14-19	<code>Setl_date</code> (дата приобретения)	Дата приобретения ценной бумаги, которая не может быть выходным или праздничным днем. Она должна предшествовать сроку выкупа.

Перед этими строками располагаются строки с датами праздников (по одной дате в каждой строке). Праздничные дни представлены в формате ГГММДД, а в самой первой строке указывается общее число праздников в году. Признаком конца потока данных служит строка с отрицательным годовым процентом.

### Программа ACCR\_INTEREST

```

procedure ACCR_INTEREST is
  -- Некоторые из употребляемых здесь типов
  -- и констант были ранее использованы в
  -- программе DATE_CONVERSION из гл.2.
  type DAY is
    (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
     FRIDAY, SATURDAY, SUNDAY);
  subtype WEEKEND is DAY range SATURDAY..SUNDAY;
  type DAY_INT is range 1 .. 31;
  type JULIAN_DAYS is range 1 .. 366;
  type MONTH is
    (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
     JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER,
     DECEMBER);
  type YEAR is range 0 .. 2050;
  type MONTH_INT is range 1 .. 12;
  package DAY_IO is new ENUMERATION_IO(DAY);
  use DAY_IO;
  package MONTH_IO is new ENUMERATION_IO(MONTH);
  use MONTH_IO;
  package YEAR_IO is new INTEGER_IO(YEAR);
  use YEAR_IO;
  package MONTH_INT is new INTEGER_IO(MONTH_INT);
  use MONTH_INT;
  package DAY_INT_IO is new INTEGER_IO(DAY_INT);
  use DAY_INT_IO;
  type DATE is
    record
      WEEK_DAY : DAY;
      MONTH_NAME : MONTH;
      MONTH_NO : MONTH_INT;
      DAY_NO : DAY_INT;
      YTD_DAYS : JULIAN_DAYS;
      TOTAL_DAYS : INTEGER;
      YEAR_NO : YEAR;
    end record;
  BASE_DATE : constant DATE :=
    (MONDAY, JANUARY, 1, 1, 1, 1, 1984);
  -- В данной программе предполагается, что во
  -- входных данных указываются только даты,
  -- следующие после 1 января 1984г.
  -- Если потребуются употребить более ранние
  -- даты, то следует изменить эту константу.
  BASE_LEAP : constant INTEGER :=
    INTEGER(BASE_DATE.YEAR_NO) / 4 +
    INTEGER(BASE_DATE.YEAR_NO) / 400 -
    INTEGER(BASE_DATE.YEAR_NO) / 100;
  -- Эта константа равна количеству високосных
  -- лет в период от 0-го года до года
  -- BASE_DATE.YEAR_NO.

```

```

type DAYS_IN_MONTH is array(MONTH_INT,BOOLEAN)
                        of DAY_INT;
ACTUAL_DAYS_IN_YEAR : constant DAYS_IN_MONTH :=
  ( (31,31), (28,29), (31,31), (30,30),
    (31,31), (30,30), (31,31), (31,31),
    (30,30), (31,31), (30,30), (31,31) );
-- Это - об'явление массива-константы, индек-
-- сированного значениями типа MONTH_INT, ле-
-- жщими в пределах от 1 до 12, и логичес-
-- кими значениями (TRUE и FALSE), причем
-- FALSE здесь обозначает, что год - не ви-
-- сокосный.
type RATE is digits 13;
ACCRUED_INT : RATE;
type COUPON_SECURITY is
  record
    COUP_DATE : RATE;
    MAT_DATE : DATE;
    SETL_DATE : DATE;
  end record;
ACT_SECUR : COUPON_SECURITY;
NEXT_DATE, PREV_COUP_DATE : DATE;
package RATE_IO is new FLOAT_IO(RATE);
use RATE_IO;
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
LEAP_YEAR : BOOLEAN;
NOFDAYS, INTERVDAYS : NATURAL;
INPUT_LEAP : INTEGER;
NO_OF_HOLIDAYS, NO_GOOD_HOLIDAYS : NATURAL;
type HOLIDAYS is array (1 .. NO_OF_HOLIDAYS)
                        of DATE;
ACT_HOLIDAYS : HOLIDAYS;
GOOD_MAT_DATE, GOOD_SETL_DATE : BOOLEAN;
GOOD_HOLIDAY, GOOD_NEXT_DATE : BOOLEAN;
-- Обратите внимание, что основой функции
-- IS_VALID_DATE и процедуры FILL_IN_DATE
-- служит текст процедуры
-- COUNT_DAYS_AND_CHECK из программы
-- DAYS_CONVERSION (см.гл.2). Кроме того,
-- заметьте, что об'явления, приведенные
-- выше, известны (видимы) в каждой из
-- следующих ниже подпрограмм.
function IS_VALID_DATE (FORM_DATE : DATE)
  return BOOLEAN is
-- Эта функция дает значение TRUE, если
-- дата, представленная переменными
-- FORM_DATE.MONTH_NO, FORM_DATE.DAY_NO
-- и FORM_DATE.YEAR_NO, правильна. В про-
-- тивном случае вырабатывается значение
-- FALSE. У этой функции имеется всего
-- лишь один формальный параметр комбини-
-- рованного типа.
begin

```

```

-- Правильно ли указан год? Если нет, то
-- функция даст значение FALSE.
if FORM_DATE.YEAR_NO < BASE_DATE.YEAR_NO
then
  return FALSE;
end if;
-- Правильность месяца не проверяется.
-- Выясним, правильно ли задано число.
-- Если нет, то функция выработает значение FALSE. Но вначале определим, не является ли год високосным.
if FORM_DATE.YEAR_NO rem 4 = 0 and
  FORM_DATE.YEAR_NO rem 100 /= 0 or
  FORM_DATE.YEAR_NO rem 400 = 0
then
  LEAP_YEAR := TRUE;
else
  LEAP_YEAR := FALSE;
end if;
-- Теперь можно использовать переменную
-- LEAP_YEAR для обращения к нужному ряду
-- массива ACTUAL_DAYS_IN_YEAR.
if FORM_DATE.YEAR_NO >
  ACTUAL_DAYS_IN_YEAR(FORM_DATE.MONTH_NO,
    LEAP_YEAR);
then
  return FALSE;
else
  return TRUE;
end if;
end IS_VALID_DATE;
procedure FILL_IN_DATE
  (PROC_F_DATE : in out DATE;
   GOOD_DATE   : in out BOOLEAN) is
-- Эта процедура вычисляет значения некоторых
-- компонент структуры PROC_F_DATE, причем
-- предполагается, что значения остальных
-- компонент -- FORM_DATE.MONTH_NO,
-- FORM_DATE.DAY_NO, FORM_DATE.YEAR_NO образуют правильную дату. Если это не так, то
-- переменная GOOD_DATE получает значение
-- FALSE.
-- В формальной части данной процедуры имеются
-- два формальных параметра: PROC_F_DATE компонента бинированного типа и GOOD_DATE логического типа.
begin
-- Если год задан в форме ГГ, то он преобразуется к представлению в виде четырех цифр.
if PROC_F_DATE.YEAR_NO < 50
then
  PROC_F_DATE.YEAR_NO :=
    PROC_F_DATE.YEAR_NO + 2000;
elsif PROC_F_DATE.YEAR_NO < 99
then
  PROC_F_DATE.YEAR_NO :=
    PROC_F_DATE.YEAR_NO + 1900;

```

```

else
  NULL;
end if;
GOOD_DATE := IS_VALID_DATE (PROC_F_DATE);
-- Определим значение переменной
-- YTD_DAYS для этой даты.
if GOOD_DATE
then
  PROC_F_DATE.YTD_DAYS :=
    JULIAN(PROC_F_DATE.DAY_NO);
  if PROC_F_DATE.MONTH_NO > 1
  then
    for I in 1 .. PROC_F_DATE.MONTH_NO - 1
    loop
      PROC_F_DATE.YTD_DAYS :=
        PROC_F_DATE.YTD_DAYS +
        JULIAN(ACTUAL_DAYS_IN_YEAR
          (I, LEAP_YEAR));
    end loop;
  end if;
end if;
-- Теперь определим, сколько дней прошло от
-- даты отсчета до представленной даты.
-- Вначале, однако, вычислим количество
-- прошедших високосных лет.
INPUT_LEAP := INTEGER(PROC_F_DATE.YEAR_NO)/4+
  INTEGER(PROC_F_DATE.YEAR_NO)/400-
  INTEGER(PROC_F_DATE.YEAR_NO)/100;
PROC_F_DATE.TOTAL_DAYS := 365 *
  INTEGER(PROC_F_DATE.YEAR_NO -
    BASE_DATE.YEAR_NO) +
  INTEGER(PROC_F_DATE.YTD_DAYS) +
  INPUT_LEAP - BASE_LEAP;
-- Найдем день недели и название месяца.
PROC_F_DATE.MONTH_NAME :=
  MONTH_VAL(PROC_F_DATE.MONTH_NO);
-- Вспомните, что атрибут VAL применим для
-- перечисляемых типов.
PROC_F_DATE.WEEK_DAY := DAY_VAL(
  (PROC_F_DATE.TOTAL_DAYS +
    DAY_POS(BASE_DATE.WEEK_DAY) - 2)
  mod 7 + 1);
-- Это выражение - довольно сложное. Вначале
-- вычисляется позиция дня недели для даты
-- отсчета (например, для вторника она будет
-- равна 2), затем она добавляется к общему
-- количеству дней и из полученного числа
-- вычитается 2. Остаток от деления итогового
-- числа на 7 дает относительную позицию
-- дня недели для входной даты. Это будет
-- целое число от 1 до 7. Для лучшего понимания
-- данного выражения попробуйте вы-
-- числить его вручную для нескольких
-- значений дат.
end FILL_IN_DATE;
function FIND_COUP_DATE(
  FORM_MAT_DATE, FORM_SETL_DATE : DATE)
return DATE is

```

```

-- Эта функция вычисляет значение комбиниро-
-- ванного типа. Выплата денег по купонам
-- производится два раза в год; число и месяц
-- первой ежегодной даты выплаты совпадают с
-- числом и месяцем даты выкупа ценной бума-
-- ги, а вторая дата выплаты на 6 месяцев
-- позже первой.
WORK_DATE_1, WORK_DATE_2 : DATE;
GOOD_WORK_DATE : BOOLEAN;
begin
  WORK_DATE_1 := FORM_MAT_DATE;
  WORK_DATE_1.YEAR_NO := FORM_SETL_DATE.YEAR_NO;
  FILL_IN_DATE(WORK_DATE_1,GOOD_WORK_DATE);
  -- Может возникнуть вопрос, почему в качестве
  -- фактического параметра процедуры
  -- FILL_IN_DATE не используется FORM_MAT_DATE?
  -- Дело в том, что соответствующий формальный
  -- параметр этой процедуры имеет вид связи
  -- in out, что подразумевает возможность
  -- изменения FORM_MAT_DATE, выступающего в
  -- роли формального параметра функции
  -- FIND_COUP_DATE, а это было бы недопустимым.
  WORK_DATE_2 := WORK_DATE_1;
  -- Переменные WORK_DATE_1 и WORK_DATE_2
  -- будут содержать две даты выплаты по купо-
  -- нам, расположенные наиболее близко к дате
  -- покупки ценных бумаг. Вычислим дату выпла-
  -- ты, непосредственно предшествующую дате
  -- покупки.
  if WORK_DATE_1.YTD_DAYS <
    FORM_SETL_DATE.YTD_DAYS
  then
    if WORK_DATE_1.MONTH_NO < 7
    then
      WORK_DATE_2.MONTH_NO :=
        WORK_DATE_2.MONTH_NO + 6;
    else
      WORK_DATE_2.MONTH_NO :=
        WORK_DATE_2.MONTH_NO - 6;
      WORK_DATE_2.YEAR_NO :=
        WORK_DATE_2.YEAR_NO + 1;
    end if;
  else
    if WORK_DATE_1.MONTH_NO > 6
    then
      WORK_DATE_2.MONTH_NO :=
        WORK_DATE_2.MONTH_NO - 6;
    else
      WORK_DATE_2.MONTH_NO :=
        WORK_DATE_2.MONTH_NO + 6;
      WORK_DATE_2.YEAR_NO :=
        WORK_DATE_2.YEAR_NO - 1;
    end if;
  end if;
  FILL_IN_DATE(WORK_DATE_2,GOOD_WORK_DATE);
  if WORK_DATE_1.TOTAL_DAYS <
    WORK_DATE_2.TOTAL_DAYS

```

```

then
  if FORM_SETL_DATE.TOTAL_DAYS >
    WORK_DATE_2.TOTAL_DAYS
  then
    return WORK_DATE_2;
  else
    return WORK_DATE_1;
  end if;
else
  if FORM_SETL_DATE.TOTAL_DAYS >
    WORK_DATE_1.TOTAL_DAYS
  then
    return WORK_DATE_1;
  else
    return WORK_DATE_2;
  end if;
end if;
end FIND_COUP_DATE;
-- Обратите внимание, что тело функции
-- IS_VALID_DATE предшествует телу процедуры
-- FILL_IN_DATE, где эта функция использует-
-- ся. Поэтому нет необходимости в об'явлении
-- функции IS_VALID_DATE. Аналогичная ситуа-
-- ция наблюдается и в случае процедуры
-- FIND_COUP_DATE: в этой процедуре использу-
-- ется процедура FILL_IN_DATE, которая не
-- нуждается в особом об'явлении, так как
-- ее тело расположено раньше, чем тело
-- процедуры FIND_COUP_DATE.
begin
  GET(NO_OF_HOLIDAYS);
  -- Далее, запишем все даты праздников в мас-
  -- сив.
  NO_GOOD_HOLIDAYS := 0;
  for I in 1 .. NO_OF_HOLIDAYS
  loop
    NO_GOOD_HOLIDAYS := NO_GOOD_HOLIDAYS + 1;
    GET(ACT_HOLIDAYS(NO_GOOD_HOLIDAYS).YEAR_NO,2);
    GET(ACT_HOLIDAYS(NO_GOOD_HOLIDAYS).MONTH_NO,2);
    GET(ACT_HOLIDAYS(NO_GOOD_HOLIDAYS).DAY_NO,2);
    FILL_IN_DATE(ACT_HOLIDAYS(NO_GOOD_HOLIDAYS),
      GOOD_HOLIDAY);
    -- Процедура FILL_IN_DATE здесь имеет
    -- формальные параметры
    -- ACT_HOLIDAYS(NO_GOOD_HOLIDAYS) и
    -- GOOD_HOLIDAY.
    if not GOOD_HOLIDAY
    then
      -- В таблицу будут помещены только пра-
      -- вильные значения дат.
      NO_GOOD_HOLIDAYS := NO_GOOD_HOLIDAYS - 1;
    end if;
    SKIP_LINE;
  end loop;
  -- Считываем входные строки до тех пор, пока
  -- не будет прочитано отрицательное значение
  -- процентов выплаты по купону.

```

```

GET(ACT_SECUR.COUP_RATE, 7);
while ACT_SECUR.COUP_RATE <= 0.0
  loop
    GET(ACT_SECUR.MAT_DATE.YEAR_NO, 2);
    GET(ACT_SECUR.MAT_DATE.MONTH_NO, 2);
    GET(ACT_SECUR.MAT_DATE.DAY_NO, 2);
    GET(ACT_SECUR.SETL_DATE.YEAR_NO, 2);
    GET(ACT_SECUR.SETL_DATE.MONTH_NO, 2);
    GET(ACT_SECUR.SETL_DATE.DAY_NO, 2);
    FILL_IN_DATE(ACT_SECUR.MAT_DATE,
      GOOD_MAT_DATE);
    -- Процедура FILL_IN_DATE вызывается с
    -- фактическими параметрами
    -- ACT_SECUR.MAT_DATE и GOOD_MAT_DATE.
    FILL_IN_DATE(ACT_SECUR.SETL_DATE,
      GOOD_SETL_DATE);
    -- Процедура FILL_IN_DATE вызывается с
    -- фактическими параметрами
    -- ACT_SECUR.SETL_DATE и GOOD_SETL_DATE.
    -- Дата приобретения ценных бумаг не должна
    -- быть больше даты выкупа. Если же это
    -- происходит, то переменная GOOD_MAT_DATE
    -- получает значение FALSE.
    if ACT_SECUR.MAT_DATE.TOTAL_DAYS <=
      ACT_SECUR.SETL_DATE.TOTAL_DAYS
    then
      GOOD_MAT_DATE := FALSE;
    end if;
    -- В выходные дни нельзя приобретать ценные
    -- бумаги. Если обнаружено, что дата приоб-
    -- ретения попадает на выходные дни, то
    -- значение переменной GOOD_SETL_DATE уста-
    -- навливается равным FALSE.
    if ACT_SECUR.SETL_DATE.WEEK_DAY in WEEKEND
    then
      GOOD_SETL_DATE := FALSE;
    end if;
    -- По праздникам также нельзя приобретать
    -- ценные бумаги. Если дата приобретения
    -- попадает на праздник, то переменная
    -- GOOD_SETL_DATE получает значение FALSE.
    for I in 1 .. NO_GOOD_HOLIDAYS
    loop
      exit when not GOOD_SETL_DATE;
      if ACT_HOLIDAYS(I).TOTAL_DAYS =
        ACT_SECUR.SETL_DATE.TOTAL_DAYS
      then
        GOOD_SETL_DATE := FALSE;
      end if;
    end loop;
    if GOOD_MAT_DATE and GOOD_SETL_DATE
    then
      -- Найти предыдущий день выплаты
      -- денег по купону.
      PREV_COUP_DATE := FIND_COUP_DATE(
        ACT_SECUR.MAT_DATE,
        ACT_SECUR.SETL_DATE);

```



```

-- Здесь вызывается функция
-- FIND_COUP_DATE с фактическими парамет-
-- рами ACT_SECUR.MAT_DATE и
-- ACT_SECUR.SETL_DATE. Она вырабатывает
-- значение типа DATE. Здесь можно было
-- бы употребить и префиксную форму за-
-- писи, например:
-- FIND_COUP_DATE(ACT_SECUR.MAT_DATE,
-- ACT_SECUR.SETL_DATE).MONTH_NO или
-- FIND_COUP_DATE(ACT_SECUR.MAT_DATE,
-- ACT_SECUR.SETL_DATE).DAY_NO.
NEXT_DATE.DAY_NO := PREV_COUP_DATE.DAY_NO;
-- Найдем дату следующего платежа по
-- купону. Здесь лучше бы было написать
-- процедуру, которая вычисляла бы сразу
-- и PREV_COUP_DATE, и NEXT_DATE. Пос-
-- мотрите упр.3 в конце данной главы.
if PREV_COUP_DATE.MONTH_NO < 7
then
  NEXT_DATE.MONTH_NO :=
    PREV_COUP_DATE.MONTH_NO + 6;
  NEXT_DATE.YEAR_NO :=
    PREV_COUP_DATE.YEAR_NO;
else
  NEXT_DATE.MONTH_NO :=
    PREV_COUP_DATE.MONTH_NO - 6;
  NEXT_DATE.YEAR_NO :=
    PREV_COUP_DATE.YEAR_NO + 1;
end if;
FILL_IN_DATE(NEXT_DATE,GOOD_NEXT_DATE);
-- Вычислим наросшие проценты.
NOFDAYS := ACT_SECUR.SETL_DATE.TOTAL_DAYS
  - PREV_COUP_DATE.TOTAL_DAYS;
INTERVDAYS := NEXT_DATE.TOTAL_DAYS
  - ACT_SECUR.SETL_DATE.TOTAL_DAYS;
ACCRUED_INT := ACT_SECUR.COUP_RATE / 2.0 *
  RATE(NOFDAYS) / RATE(INTERVDAYS);
PUT(" The accr. interest for the security"
  & " maturing ");
PUT(ACT_SECUR.MAT_DATE.YEAR_NO, 2);
PUT(ACT_SECUR.MAT_DATE.MONTH_NO, 2);
PUT(ACT_SECUR.MAT_DATE.DAY_NO, 2);
NEW_LINE;
PUT(" is ");
PUT(ACCRUED_INT, 8,2);
NEW_LINE;
else
  -- Обнаружены неправильные даты.
  PUT(" Wrong maturity or settlement date");
end if;
SKIP_LINE;
GET(ACT_SECUR.COUP_RATE, 7);
end loop;
end ACCR_INTEREST;

```

## 5.4. ПЕРЕКРЫТИЕ ПОДПРОГРАММ

В языке Ада один и тот же идентификатор можно употреблять для обозначения различных подпрограмм. В этом случае идентификатор называется *перекрытым*. Разумеется, при любом вызове перекрытой подпрограммы должна существовать возможность среди множества подпрограмм выбрать нужную. В противном случае вызов будет *неоднозначным*.

Мы уже пользовались перекрытыми подпрограммами, например, каждый раз, когда обращались к подпрограммам GET и PUT. Мы употребляли один и тот же идентификатор GET (или соответственно PUT) для чтения (или записи) строк символов и значений целого, плавающего и перечисляемого типов.

Ниже дан список спецификаций подпрограмм GET и PUT, составляющих часть родового пакета TEXT\_IO. Пометим для удобства эти процедуры как Proc\_1...Proc\_7.

```
procedure GET(ITEM: out CHARACTER); -- Proc_1
procedure GET(ITEM: out STRING);   -- Proc_2
procedure PUT(ITEM: in CHARACTER);  -- Proc_3
procedure PUT(ITEM: in STRING);     -- Proc_4
```

Для пакета INTEGER\_IO, где NUM принадлежит к целому типу, а FIELD (поле) имеет натуральный целый подтип, мы использовали спецификацию:

```
procedure GET (ITEM : out NUM;
              WIDTH : in FIELD := 0); -- Proc_5
```

В пакете FLOAT\_IO (в спецификациях подпрограмм ввода-вывода) используются инициализированные переменные целого типа и переменная NUM типа FLOAT. Переменная NUM здесь отличается по типу от одноименной переменной из пакета INTEGER\_IO. Ниже приведены некоторые из спецификаций процедур PUT и GET:

```
DEFAULT_FORE : FIELD := 2;
DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
DEFAULT_EXP  : FIELD := 3;
procedure PUT(ITEM : in NUM;
              FORE : in FIELD := DEFAULT_FORE;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP;
              -- Proc_6

procedure GET(ITEM : out NUM;
              WIDTH: in FIELD := 0); -- Proc_7
```

**Пример.** Проиллюстрируем применение перекрытых процедур. Пусть имеются такие объявления переменных:

```
I : INTEGER;
CH : CHARACTER;
F : FLOAT;
STR : STRING;
```

Ниже приведены вызовы процедур. Предполагается, что все эти процедуры доступны.

Вызов	Описание
PUT('A');	Вызывается процедура Proc_3. Константа 'A' принадлежит к типу CHARACTER, а эта процедура — единственная из процедур с именем PUT, в которой формальный параметр имеет тип CHARACTER

GET(CH);	Вызывается процедура Proc_1. Переменная CH имеет тип CHARACTER, а Proc_1 – единственная из процедур с именем GET, в которой формальный параметр относится к типу CHARACTER
GET(STR);	Вызывается процедура Proc_2. Переменная STR принадлежит к типу STRING, а Proc_2 – единственная из процедур с именем GET, в которой формальный параметр имеет тип STRING
PUT("Tomorrow");	Вызывается процедура Proc_4. Константа Tomorrow (Завтра) относится к типу STRING, а Proc_4 – единственная из процедур с именем PUT, в которой формальный параметр имеет тип STRING
GET(I)	Вызывается процедура Proc_5. Тип переменной I – INTEGER, а Proc_5 – единственная среди процедур с именем GET, в которой формальный параметр имеет тип INTEGER. Второй аргумент отсутствует, и ширина считываемого поля получает значение по умолчанию
PUT(F, FORE => 7, AFT => 3, EXP => 3);	Вызывается процедура Proc_6. Тип F – FLOAT, а Proc_6 – единственная среди процедур с именем PUT, обладающая формальным параметром типа FLOAT. Здесь применена форма записи с поименованными компонентами
GET(F, 7);	Вызывается процедура Proc_7. Тип переменной F – FLOAT, а Proc_7 – единственная из процедур с именем GET, имеющая формальный параметр типа FLOAT

Перекрытие подпрограмм будет восприниматься вполне естественно, если употреблять имена, несущие определенный смысл для программиста. Более того, в качестве имен функций можно использовать названия операций, например: +, –, \*, / и \*\*. Их можно применить, скажем, для типов, у которых эти операции отсутствовали. Это оправдано в тех случаях, когда способ определения новых операций будет напоминать разработчику об их первоначальном значении. Например, в языке Ада отсутствует операция возведения плавающего числа в степень, представленную также плавающим числом. Можно написать функцию, обозначаемую как \*\*, которая перекроет операцию \*\*. Формальная часть такой функции будет содержать два параметра типа FLOAT. Можно также определить, например, новый тип COMPLEX (комплексный), который будет предназначен для моделирования свойств комплексных чисел. Для этого нового типа можно ввести операцию умножения, которая перекроет символ имеющейся операции \*. Умножение матриц – это еще один пример возможного перекрытия символа операции \*.

**Пример.** Здесь приводится пример перекрытия операций для типа DATE (Дата), использовавшегося в программе ACCR\_INTEREST. Предполагается, что тело функции IS\_VALID\_DATE (Проверка правильности даты), тело процедуры FILL\_IN\_DATE (Заполнить дату) и декларативная часть процедуры ACCR\_INTEREST предшествуют описанным здесь функциям. Объединение всех этих подпрограмм в один пакет будет выполнено в гл. 7.

```

function "<=" (X,Y : DATE := BASE_DATE)
    return BOOLEAN is
-- Здесь считается, что дата состоит из следующих
-- компонент: MONTH_NO, YEAR_NO и DAY_NO
-- (месяц, год и день).
    LOCAL_GOOD_X, LOCAL_GOOD_Y : BOOLEAN;
    LOCAL_X, LOCAL_Y : DATE;
begin
    LOCAL_X := X;    LOCAL_Y := Y;
-- Переменные X и Y - это входные формальные
-- параметры. Их нельзя использовать при обра-
-- щении к FILL_IN_DATE, так как значения вхо-
-- дящих в них компонент могут быть изменены.
    FILL_IN_DATE (LOCAL_X, LOCAL_GOOD_X);
    FILL_IN_DATE (LOCAL_Y, LOCAL_GOOD_Y);
    if LOCAL_GOOD_X and LOCAL_GOOD_Y and
        LOCAL_X.TOTAL_DAYS <= LOCAL_Y.TOTAL_DAYS
    then
        return TRUE;
    else
        return FALSE;
    end if;
-- Эту же функцию можно было бы реализовать
-- просто с помощью сравнения соответствующих
-- компонент даты, например :
-- if X.YEAR_NO < Y.YEAR_NO or
--    X.YEAR_NO = Y.YEAR_NO and
--    X.MONTH_NO < Y.MONTH_NO or
--    X.YEAR_NO = Y.YEAR_NO and
--    X.MONTH_NO = Y.MONTH_NO and
--    X.DAY_NO <= Y.DAY_NO
-- then return TRUE;
-- else return FALSE;
-- end if;
-- В первом варианте, однако, выполняется до-
-- полнительная проверка правильности даты.
end "<=";
-- В приведенной ниже функции предполагается, что
-- заданный вторым операндом временной интервал
-- (год, месяц, день) - YEAR_NO, MONTH_NO и DAY_NO -
-- отсчитывается относительно первого операнда
-- (т.е. X), а не от нулевого года как
-- остальные даты.
function "+" (X,Y : DATE)
    return DATE is
    LOCAL_GOOD_Z, LEAP_Z : BOOLEAN;
    LOCAL_Z : DATE;
begin
    LOCAL_Z.DAY_NO := X.DAY_NO + Y.DAY_NO;
    LOCAL_Z.MONTH_NO := X.MONTH_NO + Y.MONTH_NO;
    LOCAL_Z.YEAR_NO := X.YEAR_NO + Y.YEAR_NO;
    if LOCAL_Z.MONTH_NO > 12
    then
        LOCAL_Z.YEAR_NO := LOCAL_Z.YEAR_NO + 1;
        LOCAL_Z.MONTH_NO := LOCAL_Z.MONTH_NO - 12;
    end if;

```

```

-- Проверить, не является ли полученный год
-- високосным.
if LOCAL_Z.YEAR_NO rem 4 = 0 and
   LOCAL_Z.YEAR_NO rem 100 /= 0 and
   LOCAL_Z.YEAR_NO rem 400 = 0
then
  LEAP_Z := TRUE;
else
  LEAP_Z := FALSE;
end if;
-- Теперь для обращения к нужной строке массива
-- ACTUAL_DAYS_IN_YEAR можно использовать
-- переменную LEAP_Z.
if LOCAL_Z.DAY_NO >
  ACTUAL_DAYS_IN_YEAR(LOCAL_Z.MONTH_NO, LEAP_Z)
then
  LOCAL_Z.DAY_NO := LOCAL_Z.DAY_NO -
    ACTUAL_DAYS_IN_YEAR(LOCAL_Z.MONTH_NO, LEAP_Z);
  LOCAL_Z.MONTH_NO := LOCAL_Z.MONTH_NO + 1;
  if LOCAL_Z.MONTH_NO > 12
  then
    LOCAL_Z.YEAR_NO := LOCAL_Z.YEAR_NO + 1;
    LOCAL_Z.MONTH_NO := LOCAL_Z.MONTH_NO - 12;
  end if;
end if;
FILL_IN_DATE (LOCAL_Z, LOCAL_GOOD_Z);
return LOCAL_Z;
end "+";

```

Пусть эти функции составляют часть объявлений подпрограммы, содержащей также следующие объявления:

```
ACT_X, ACT_Y, ACT_Z : DATE;
```

и каждая из этих переменных имеет корректное значение даты. Тогда можно записать:

```

if ACT_X <= ACT_Y
then
  ACT_X := ACT_X + ACT_Z;
else
  ACT_Y := ACT_Y + ACT_Z;
end if;

```

Здесь операции "<=" и "+", в которых участвуют переменные ACT\_X, ACT\_Y и ACT\_Z, перекрыты. Транслятор сможет подставить верную функцию после проверки типов операндов.

## 5.5. РЕКУРСИВНЫЕ ВЫЗОВЫ ПОДПРОГРАММ

Функции и подпрограммы в языке Ада могут быть *рекурсивными*. Это означает, что они могут вызывать сами себя. Рекурсия является мощным средством языка, позволяющим зачастую сгенерировать и выполнить большое количество операторов при записи относительно небольшого числа строк программы. Однако проследить за работой рекурсивных программ будет непростой задачей. Приведем несколько несложных примеров рекурсивных программ. Один из них — функция, вычисляющая факториал положительного целого числа.

```

function FACTORIAL ( I : NATURAL )
    return NATURAL is
begin
    if I = 0
    then
        return 1;
    else
        return I * FACTORIAL(I-1);
        -- Здесь функция FACTORIAL вызывает сама
        -- себя, фактическим параметром является I-1.
    end if;
end FACTORIAL;

```

Если переменная *I* имеет тип NATURAL (Натуральный), то обратиться к функции FACTORIAL можно следующим образом:

```
I := FACTORIAL (5);
```

В результате получится число  $120 = 5 \times 4 \times 3 \times 2 \times 1$ . Вначале происходит вызов функции FACTORIAL с фактическим параметром 5. Этот параметр — входной, как и вообще все параметры функций. Функция FACTORIAL выполняется с начальным значением *I*, равным 5. Затем управление попадает на оператор

```
return I * FACTORIAL (I - 1);
```

Он осуществляет еще одно обращение к функции FACTORIAL, на этот раз с фактическим параметром  $I - 1 = 4$ .

Обратите внимание, что вызов со значением 5 пока не завершился, так как выражение  $I * \text{FACTORIAL}(I - 1)$  еще не вычислено. Что произойдет с незавершенным вызовом с фактическим параметром 5 при выполнении вызова с фактическим параметром, равным 4? Среда незавершенного вызова сохранится, т.е. будет запомнен адрес точки прерывания, будут сохранены значения переменных, параметров и т.д. Сохранение среды для незавершенных вызовов производится по принципу стека, т.е. «первым пришел — последним вышел». Оно будет выполняться до тех пор, пока не станет возможным непосредственное вычисление выражения  $I * \text{FACTORIAL}(I - 1)$ , т.е. пока функция FACTORIAL ( $I - 1$ ) не выработает значение.

Если мы проследим далее за выполнением этой функции, то увидим, что среды вызовов для фактических параметров, равных 4, 3, 2 и 1, будут сохранены, поскольку для вычисления выражения  $I * \text{FACTORIAL}(I - 1)$  требуется конкретное значение функции FACTORIAL ( $I - 1$ ). Когда, наконец, произойдет обращение к FACTORIAL (0), в системе будут уже храниться данные о пяти незавершенных вызовах. Верхнее положение в стеке занимает FACTORIAL (1) (последний незавершенный вызов), а нижнее — FACTORIAL (5) (самый первый вызов). После завершения вызова FACTORIAL (0) вырабатывается значение, равное 1. Затем в свою очередь заканчивается выполнение FACTORIAL (1). Этот незавершенный вызов был самым последним в стеке, поэтому он выполняется в первую очередь. В результате получается 1. После этого становится возможным выполнение FACTORIAL (2), что даст 2. Потом последовательно выполняются вызовы FACTORIAL (3), FACTORIAL (4) и, в конце концов, FACTORIAL (5). В итоге получится 120.

Все подпрограммы языка Ады *реентерабельны*. Значение термина «реентерабельный» легче объяснить по принципу «от противного». Подпрограмма, которая ведет себя по-разному в разные промежутки времени при вызовах с одинаковыми значениями фактических параметров и переменных, известных за ее пределами, не является реентерабельной. Предположим, например, что имеется подпрограмма, которая при первом вызове присваивает некоторым своим локальным переменным значения, известные только внутри ее. (В противоположность нелокальным переменным, кото-

рые могут быть известны также и вне этой подпрограммы.) Благодаря установленным значениям этих локальных переменных, при втором вызове подпрограмма будет выполняться в иной логической последовательности, чем при первом обращении. Эта подпрограмма нересентерабельна.

В языке Ада не допускается такое поведение подпрограмм, поскольку обработка декларативной части подпрограмм выполняется при каждом вызове заново. Заметьте, однако, что каждая такая «обработка» требует дополнительных затрат времени на вызов подпрограммы. Среди прочих преимуществ свойство ресентерабельности предоставляет, например, возможность использовать в нескольких программах на языке Ада только одну копию нужной подпрограммы. При этом все программы могут одновременно обращаться к одной и той же подпрограмме.

Теперь продемонстрируем использование рекурсивных подпрограмм как мощного выразительного средства программирования, а также их связь с рекурсивными определениями ссылочных типов. Здесь будет приведена модификация программы ACCESS\_GRADES из гл. 3. Входные данные программы те же, что и раньше. Как и в программе ACCESS\_GRADES, информация, необходимая для оценки каждой контрольной работы, помещается в паре строк следующего формата. В первой строке имеются такие поля:

Позиции	Данные
1–5	Название (ID) предмета, например MATH1 (математика № 1) или ENGL5 (английский язык № 5)
6–7	Число вопросов в контрольной работе (от 20 до 50)

Во второй строке располагаются ключи ответов. Ключ – это последовательность цифр от 1 до 5, показывающая номера правильных вариантов ответов. Число цифр в этой строке равно количеству вопросов, указанному в первой записи. Признаком конца последовательности строк первого и второго рода служит запись с обозначением названия предмета XXXXX.

Остальные входные строки представляют собой наборы ответов студентов. В первых десяти позициях размещается личный номер (ID) студента, а в следующих пяти позициях – название предмета. Сами ответы начинаются с шестнадцатой позиции. Признаком конца файла служит запись с номером студента, равным 999999999. Заметьте, что студент может выполнять несколько контрольных работ по разным предметам. Строки с ответами студентов следуют в произвольном порядке. Каждый студент может выполнить не более 25 разных контрольных работ, что соответствует приблизительно семи предметам за семестр. На рис. 3.7 был дан пример ключей ответов к контрольным работам, а на рис. 5.1 приводятся примеры строк с идентификаторами студентов и их ответами.

```
1234567890COMP1123451234555555111115
1111199999ENGL1333335555533333555553333
1111111111ENGL1123456789012345678901234
1188888888MATH1222222222333333333322
2222222222ENGL155555555555555555555555
9999999999
```

Рис. 5.1. Тестовые данные: пять ответов студентов.

Выходные данные модифицированной программы будут отличаться от того, что выдавала программа ACCESS\_GRADES. Для каждой вводимой строки с ответами студента строится или обновляется ведомость успеваемости этого студента путем добавления названия предмета и оценки за контрольную работу. После окончания считывания всех строк с ответами на печать выводится список сведений об успеваемости студентов. Он печатается по возрастанию номеров студентов. Каждая

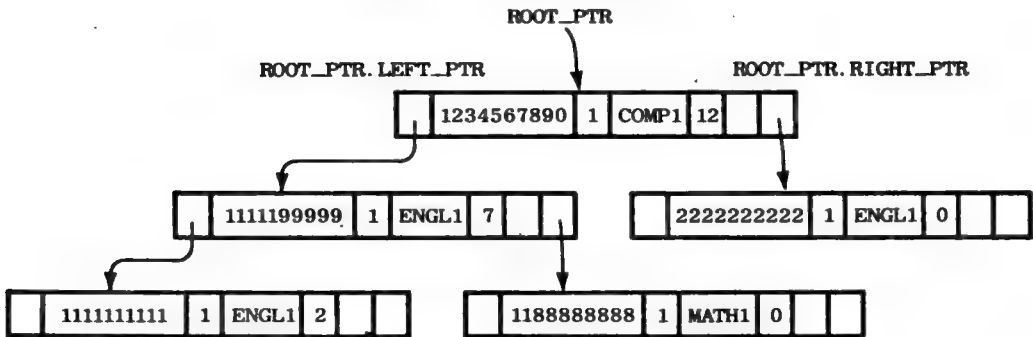


Рис. 5.2. Пример двоичного дерева.

выходная запись списка содержит информацию об одном студенте: номер студента, изучаемые им предметы, оценки за контрольные работы.

Сведения об успеваемости студентов образуют структуру данных, называемую *двоичным деревом*. Организация двоичного дерева показана на рис. 5.2. На нем изображен окончательный вид дерева для входных данных, приведенных на рис. 5.1.

Сведения об успеваемости студентов мы идентифицируем по номерам студентов, считая при этом, что у каждого студента есть свой неповторяющийся личный номер. Одно из преимуществ организации данных в виде двоичного дерева состоит в том, что если номера студентов представляют собой случайную величину с равномерным законом распределения, то добавление, выборка и обновление сведений об успеваемости осуществляются достаточно просто и быстро.

### Программа RECUR\_PROC\_GRADES

```

with TEXT_IO; use TEXT_IO;
procedure RECUR_PROC_GRADES is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type CHOICES is range 1 .. 5;
  type POSSIBLE_QUESTIONS is range 20 .. 50;
  package CHO_IO is new INTEGER_IO(CHOICES);
  use CHO_IO;
  package POSS_IO is new
    INTEGER_IO(POSSIBLE_QUESTIONS);
  use POSS_IO;
  DNO_QUESTIONS : POSSIBLE_QUESTIONS;
  type ANSWERS is array ( 1 .. DNO_QUESTIONS ) of
    CHOICES;
  type TEST_KEY;
  -- Незавершенное объявление типа, такое же самое,
  -- как и в программе ASSESS_GRADES.
  type TEST_KEY_PTR is access TEST_KEY;
  type TEST_KEY is
    record
      SUBJ          : STRING(1 .. 5);
      NO_QUESTIONS : POSSIBLE_QUESTIONS;
      KEY_ANSWERS   : ANSWERS;
      NEXT_TEST     : TEST_KEY_PTR;
    end record;
  CURR_TEST, START_TEST : TEST_KEY_PTR;

```



```

GOOD_ANSWERS : INTEGER ;
type IN_REC is
  record
    STUDENT_ID : STRING ( 1 .. 10 );
    SUBJECT_ID : STRING ( 1 .. 5 );
    STUDENT_ANSWERS : ANSWERS;
  end record;
CURR_REC : IN_REC;
-- До этого места текст данной программы
-- совпадает с текстом программы
-- ACCESS_GRADES из гл.3. Далее добав-
-- ляются об'явления, необходимые для
-- построения двоичных деревьев.
type T_PAIR is
  -- Этот тип об'единяет сведения о
  -- предмете и оценке за него. Если
  -- название учебного курса - ENGL, то
  -- для этого предмета могут быть следую-
  -- щие названия контрольных работ:
  -- ENGL1, ENGL2 и ENGL3.
  record
    SUBJ_MAT : STRING ( 1 .. 5 );
    SCORE : NATURAL;
  end record;
CURR_PAIR : T_PAIR;
type SEMESTER_TESTS is array (1 .. 25 )
  of T_PAIR;

type BIG_REC is
  record
    -- Счетчик NO_TESTS подсчитывает количество
    -- контрольных работ у студента.
    BIG_ST_ID : STRING ( 1 .. 10 );
    NO_TESTS : NATURAL;
    ST_TESTS : SEMESTER_TESTS;
  end record;
type TREE_RECORD;
type TREE_REC_PTR is access TREE_RECORD;
type TREE_RECORD is
  record
    LEFT_PTR : TREE_REC_PTR;
    INFO : BIG_REC;
    RIGHT_PTR : TREE_REC_PTR;
  end record;
ROOT_PTR : TREE_REC_PTR;
-- Переменная, указывающая на вершину двоичного
-- дерева, - это единственная переменная, требу-
-- ющаяся для идентификации этого дерева.
procedure INSERT_OR_UPDATE_STUDENT
  ( FORM_ST_ID : STRING ;
    FORM_T_PAIR : T_PAIR ;
    CURR_TREE : in out TREE_REC_PTR ) is
  -- Эта процедура вставляет структуру,
  -- содержащую сведения о студентах, или
  -- изменяет данные в ней. На эту структу-
  -- ру указывает ссылочная переменная
  -- типа TREE_REC_PTR. CURR_TREE факти-
  -- чески будет указывать на тот иденти-

```

```

-- фактор студента, который равен
-- FORM_ST_ID. Если он не обнаружен, то
-- создается новая структура со сведения-
-- ями о студенте, которая будет проини-
-- циализирована значениями FORM_ST_ID и
-- FORM_T_PAIR.
begin
  if CURR_TREE = NULL
  -- Этот оператор дает значение TRUE только в
  -- том случае, когда значение личного номера
  -- студента, хранящееся в FORM_ST_ID, не на-
  -- ходится в дереве.
  then
    CURR_TREE := new TREE_RECORD;
    CURR_TREE.INFO.BIG_ST_ID := FORM_ST_ID;
    CURR_TREE.INFO.NO_TESTS := 1;
    CURR_TREE.INFO.ST_TESTS(1) := FORM_T_PAIR;
    -- Вспомните, что CURR_TREE.LEFT_PTR и
    -- CURR_TREE.RIGHT_PTR инициализируются значе-
    -- нием NULL самой Ада-системой. Теперь
    -- CURR_TREE.all - это новая структура с дан-
    -- ными о студенте. Для тестовых данных, при-
    -- веденных на рис.5.1, данная точка программы
    -- будет пройдена пять раз.
  elsif FORM_ST_ID < CURR_TREE.INFO.BIG_ST_ID
    -- Для указанных тестовых данных это условие
    -- будет справедливо один раз при добавлении
    -- данных для личного номера студента, равно-
    -- го 1111199999, два раза для номера
    -- 1111111111, один раз для 1188888888 и ни
    -- одного раза для номера 2222222222.
    then
      INSERT_OR_UPDATE_STUDENT (FORM_ST_ID,
        FORM_T_PAIR, CURR_TREE.LEFT_PTR );
      -- В этом операторе производится рекурсивный
      -- вызов процедуры INSERT_OR_UPDATE_STUDENTS.
      -- Текущая среда вызова сохраняется, и про-
      -- цедура вызывается повторно с фактическим
      -- параметром, равным левому указателю для
      -- CURR_TREE. Вызов завершается, после чего
      -- возможность несохранения предыдущего
      -- вызова существует только тогда, когда
      -- CURR_TREE пуст или если найден номер сту-
      -- дента, равный FORM_ST_ID.
    elsif FORM_ST_ID > CURR_TREE.INFO.BIG_ST_ID
      -- Это условие будет истинно один раз при
      -- добавлении 1188888888 и один раз при до-
      -- бавлении 2222222222.
      then
        INSERT_OR_UPDATE_STUDENT (FORM_ST_ID,
          FORM_T_PAIR, CURR_TREE.RIGHT_PTR );
        -- Этот оператор представляет собой рекур-
        -- сивное обращение к процедуре
        -- INSERT_OR_UPDATE_STUDENT. Среда текущего
        -- вызова сохранится, а процедура вызывается
        -- повторно со значением правого указателя
        -- для CURR_TREE.
      else

```

```

-- Изменить данные в структуре, содержащей
-- сведения о данном студенте. Для приве-
-- денных тестовых данных эти операторы бу-
-- дут выполнены дважды для номера
-- 1111111111 и один раз для номера
-- 2222222222.
CURR_TREE.INFO.NO_TESTS :=
    CURR_TREE.INFO.NO_TESTS + 1;
CURR_TREE.INFO.ST_TESTS (
    CURR_TREE.INFO.NO_TESTS) := FORM_T_PAIR;
end if;
end INSERT_OR_UPDATE_STUDENT;
procedure LIST_STUDENTS(FORM_TREE : TREE_REC_PTR)
is
-- Эта процедура выводит список сведений обо
-- всех студентах, информация о которых хранит-
-- ся в двоичном дереве. В самом начале зада-
-- ются данные о вершине дерева.
begin
    if FORM_TREE /= NULL
    -- Если FORM_TREE не равно NULL, то программа
    -- попытается вывести список сведений о сту-
    -- дентах, хранящихся в ветвях двоичного де-
    -- рева FORM_TREE в соответствии с перечис-
    -- ленными ниже условиями.
    then
    -- Вначале будут приведены данные из левых
    -- ветвей.
    LIST_STUDENTS ( FORM_TREE.LEFT_PTR );
    -- Эта точка программы помечается обозначе-
    -- нием **AA**.
    -- Затем будет отображена информация о сту-
    -- дентах.
    PUT ( FORM_TREE.INFO.BIG_ST_ID );
    for I in 1 .. FORM_TREE.INFO.NO_TESTS
    loop
        PUT(FORM_TREE.INFO.ST_TESTS(I).SUBJ_MAT);
        PUT(FORM_TREE.INFO.ST_TESTS(I).SCORE);
    end loop;
    -- В завершение выдаются сведения из правых
    -- ветвей.
    -- Эта точка программы помечается как **BB**.
    LIST_STUDENTS ( FORM_TREE.RIGHT_PTR );
    -- Эта точка программы помечается как **CC**.
    -- Приведенный порядок действий подразумевает,
    -- что структуры со сведениями о студентах
    -- будут выводиться в порядке возрастания
    -- личных номеров студентов.
    end if;
end LIST_STUDENTS;
begin
    CURR_TEST := new TEST_KEY;
    GET(CURR_TEST.SUBJ);
    while CURR_TEST.SUBJ /= "XXXXX"
    loop
    -- Здесь строится список контрольных работ
    -- (аналогично списку для программы
    -- ACCESS_GRADES из гл.3).

```

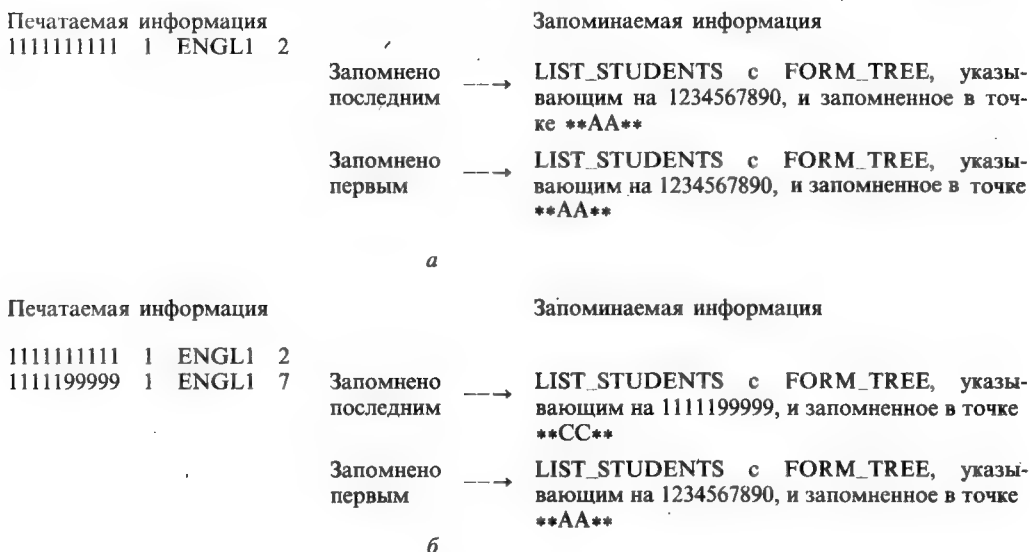
```
GET (CURR_TEST.NO_QUESTIONS,2);
DNO_QUESTIONS := CURR_TEST.NO_QUESTIONS;
SKIP_LINE;
for I in 1 .. CURR_TEST.NO_QUESTIONS
  loop
    GET(CURR_TEST.KEY_ANSWERS (I), 1);
  and loop;
  if START_TEST = NULL
    then
      -- START_TEST равно NULL только в самом
      -- начале.
      START_TEST := CURR_TEST;
    else
      CURR_TEST.NEXT_TEST := START_TEST;
      START_TEST := CURR_TEST
      -- Каждая новая контрольная работа ставится
      -- в начало списка.
    end if;
  SKIP_LINE;
  CURR_TEST := new TEST_KEY;
  GET(CURR_TEST.SUBJ);
end loop;
SKIP_LINE;
GET ( CURR_REC.STUDENT_ID );
while CURR_REC.STUDENT_ID /= "9999999999"
  loop
    GET (CURR_REC.SUBJECT_ID);
    CURR_TEST := START_TEST;
    while CURR_TEST /= NULL or
      CURR_TEST.SUBJ /= CURR_REC.SUBJECT_ID
      loop
        -- Поиск в списке предмета, по которому
        -- проводится контрольная работа.
        CURR_TEST := CURR_TEST.NEXT_TEST;
      end loop;
    if CURR_TEST.SUBJ = CURR_REC.SUBJECT_ID
      then
        -- Если результат проверки - TRUE, то предмет
        -- найден. В противном случае такого
        -- названия предмета нет в списке.
        GOOD_ANSWERS := 0;
        for J in 1 .. CURR_TEST.NO_QUESTIONS
          loop
            GET(CURR_REC.STUDENT_ANSWERS(J), 1);
            if CURR_REC.STUDENT_ANSWERS(J) =
              CURR_TEST.KEY_ANSWERS (J)
              then
                GOOD_ANSWERS := GOOD_ANSWERS + 1 ;
              end if;
          end loop;
        -- Теперь вычисляется оценка за контрольные
        -- работы. Кроме того, обновляются сведения
        -- в структуре с данными о студенте, либо
        -- если это - новый студент, то создается
        -- новая структура с данными о нем. Но зна-
        -- чение необходимо вычислить значения фак-
```

```

-- тических параметров.
CURR_PAIR.SUBJ_MAT := CURR_REC.SUBJECT_ID;
CURR_PAIR.SCORE := GOOD_ANSWERS;
INSERT_OR_UPDATE_STUDENTS (
    CURR_REC.STUDENT_ID, CURR_PAIR, ROOT_PTR);
-- Для первого студента указатель ROOT_PTR
-- имеет пустое значение перед вызовом дан-
-- ной процедуры, а после вызова ROOT_PTR
-- будет указывать на первый личный номер
-- студента, равный "1234567890". Впослед-
-- ствии значение ROOT_PTR не будет меняться.
else
    PUT(" No such subject: ");
    PUT(CURR_REC.SUBJECT_ID);
end if;
SKIP_LINE;
GET ( CURR_REC.STUDENT_ID );
end loop;
-- Обрабатываются все входные строки
-- со сведениями о студентах.
LIST_STUDENTS ( ROOT_PTR );
-- Перечисляются студенты в соответствии
-- со значениями их личных номеров, а также
-- отображаются названия предметов для конт-
-- рольных работ и оценки за эти работы.
end ACCESS_GRADES;

```

Прослеживание работы рекурсивных процедур из этой программы – трудоемкая задача, требующая внимания. Рис. 5.3,а и 5.3,б показывают, как работает рекурсивная процедура на примере выполнения процедуры LIST\_STUDENTS.



**Рис. 5.3.** Выполнение подпрограммы LIST\_STUDENTS.

а–значение переменной FORM\_TREE указывает на личный номер студента 1111111111. Точка \*\*BB\*\*;  
 б–значение переменной FORM\_TREE указывает на личный номер студента 1188888888. Точка \*\*BB\*\*.

## УПРАЖНЕНИЯ

1. Перепишите заново одну или несколько процедур программы NAME\_PHONE из гл. 4 так, чтобы в них в соответствии с нижеследующими объявлениями присутствовали формальные параметры. Заметьте, что границы строки FORM\_LINE — границы фактических параметров. Здесь рекомендуется использовать атрибуты FORM\_LINE/LAST и FORM\_LINE/LENGTH. Значения атрибутов формальных параметров определяются по значениям атрибутов фактических параметров.

```

procedure LOW_TO_UPPER_N_CT_COMMAS
  (FORM_LINE : in out STRING);
procedure IGNORE_LEADING_SPACES
  (FORM_LINE : in out STRING ;
   FORM_END_POS : out NATURAL );
  -- Здесь переменная FORM_END_POS должна
  -- давать значение первой позиции стро-
  -- ки, в которой располагается символ,
  -- отличный от пробела.
procedure FIND_NEXT_SP_OR_COMMA
  (FORM_LINE : in out STRING ;
   FORM_START_POS : in out NATURAL ;
   FORM_END_POS : in out NATURAL);
procedure PLACE_SPACES
  (FORM_LINE : in out STRING ;
   FORM_START_POS : in out NATURAL ;
   FORM_END_POS : in out NATURAL);
function IS_CORRECT_NAME
  (FORM_LINE : STRING ;
   FORM_START_POS : NATURAL ;
   FORM_END_POS : NATURAL)
  return BOOLEAN;
procedure XTR_NAME
  -- Эта процедура заменяет процедуры
  -- XTR_N_VAL_FIRST_NAME и
  -- XTR_N_VAL_LAST_NAME.
  (FORM_LINE : in out STRING ;
   FORM_START_POS : in out NATURAL ;
   FORM_END_POS : in out NATURAL ;
   XTR_NAME : out STRING);
  -- Выделенное имя помещается в
  -- XTR_NAME.
procedure XTR_N_VAL_MDL_INIT
  (FORM_LINE : in out STRING ;
   FORM_START_POS : in out NATURAL ;
   FORM_END_POS : in out NATURAL);
procedure XTR_N_VAL_PHONE
  (FORM_LINE : in out STRING ;
   FORM_START_POS : in out NATURAL ;
   FORM_END_POS : in out NATURAL ;
   XTR_PHONE : out STRING);

```

2. Напишите функцию, которая вычисляет положительное значение квадратного корня из положительного действительного числа. Вы можете использовать метод Ньютона, при котором  $(n + 1)$ -е приближение к значению квадратного корня из числа  $X$  вычисляется по формуле:

$$0.5 * (\text{Curr\_approx} + X/\text{Curr\_approx});$$

где `Curr_approx` (Текущая аппроксимация) —  $n$ -я аппроксимация значения квадратного корня. Итеративный процесс вычислений должен будет прекратиться, когда разность между значениями двух соседних аппроксимаций станет меньше некоторого заранее заданного малого положительного числа.

3. Замените функцию `FIND_COUP_DATE` из программы `ACCR_INTEREST`, помещенной в настоящей главе, на процедуру, которая будет находить дату выплаты по купонам перед днем приобретения ценной бумаги и после него. Объявление для новой процедуры может быть таким:

```
procedure GET_COUP_DATES
( FORM_MAT_DATE : DATE;
  FORM_SETL_DATE : DATE;
  FORM_PREV_DATE : out DATE;
  FORM_NEXT_DATE : out DATE );
```

Здесь `FORM_PREV_DATE` — дата платежа по купону непосредственно перед днем приобретения, а `FORM_NEXT_DATE` — ближайшая дата платежа, следующая за днем приобретения. Проведите необходимую модификацию программы `ACCR_INTEREST`.

4. Замените процедуру `CHECK_GRADE` (проверки оценки), которая входит в программу `GR_POINT_AVE` из гл. 4, на процедуру с таким объявлением:

```
procedure CHECK_GRADE
( FORM_COURSE_GR : STRING (1 .. 2);
  FORM_GR_POINT : out FLOAT;
  FORM_VALID_GR : out BOOLEAN);
```

Переменная `FORM_VALID_GR` должна выполнять роль переменной `VALID_GRADE`, а переменная `FORM_GR_POINT` должна заменить переменную `GRADE_POINT`. В связи с введением новой процедуры сделайте необходимые изменения в программе `GR_POINT_AVE`.

5. Измените программу `RECUR_PROC_GRADES` так, чтобы после считывания строк обо всех студентах и занесения данных о них в двоичное дерево каждая ведомость успеваемости студентов модифицировалась следующим образом: если студент выполняет более двух контрольных работ по одному предмету, то остаются только две наилучшие оценки, а остальные оценки по данному предмету убираются. Предметы считаются одинаковыми, если первые 4 символа их названий (переменная `SUBJ_MAT`) совпадают.

# Декларативные части и инструкции транслятору

## 6.1. ОБРАБОТКА ДЕКЛАРАТИВНЫХ ЧАСТЕЙ

Как мы уже видели, тело подпрограммы делится на декларативную часть и исполняемую часть, состоящую из последовательности операторов. В декларативной части связываются между собой имена и объявляемые ресурсы, а в исполняемой части описываются действия, которые должна выполнять подпрограмма. Порядок работы с подпрограммой таков: вначале обрабатывается декларативная часть, а затем выполняются операторы, входящие в исполняемую часть.

Обработка декларативной части производится в несколько этапов. На первом из них вводятся идентификаторы. Они вводятся в том месте, где встречаются первый раз. Это может «заслонить» другие идентичные идентификаторы, которые были обнаружены ранее. Затем только что введенные объявления ресурсов обрабатываются. К примеру, в процессе обработки объявлений вычисляются статические выражения. Для объектов устанавливается тип и вычисляются уточнения. Обработка объявлений функций и процедур включает обработку объявлений параметров, которая осуществляется в том порядке, в каком эти параметры располагаются в списке. Обработка определения типа включает установление множества значений и операций, допустимых для этих значений. Последним шагом процесса обработки объявлений является возможная инициализация объекта.

Строки, образующие декларативную часть, обрабатываются последовательно. Если в декларативной части модуля встречается объявление подпрограммы, то и тело этой подпрограммы должно быть представлено в этой же декларативной части. Исключением является случай, когда тело подпрограммы компилируется отдельно (см. гл. 9).

### 6.1.1. Область действия и видимость идентификаторов

*Область действия* идентификаторов представляет собой участок текста программы на Аде, в которой остается в силе объявление этого идентификатора. Она начинается в той точке, где идентификатор вводится в программу. Здесь предполагается, что он объявлен в декларативной части блока или подпрограммы. В подпрограммах или блоках область действия идентификатора заканчивается в месте окончания текста подпрограммы или блока, декларативная часть которого содержит этот идентификатор.

Идентификатор может быть, а может и не быть *видимым* в своей области действия. Имеется в виду то, что в некоторых участках программы, входящих в область действия данного идентификатора, этот идентификатор может быть неизвестен, если он использован сам по себе<sup>1)</sup>. Примеры такой ситуации будут приведены в дальнейшем.

Нельзя использовать никакой ресурс (объекты, числа и т.д.) до завершения обработки его объявления. По этой причине мы всегда обеспечивали расположение

<sup>1)</sup> То есть без префикса.— Прим. перев.



тела подпрограммы текстуально раньше точки вызова этой подпрограммы. По этой же самой причине в случае инициализации переменной или константы с использованием других идентификаторов мы обязательно делали так, чтобы эти идентификаторы вводились и объявления их обрабатывались ранее.

Во время обработки декларативной части может не хватить памяти ЭВМ. В этом случае возникает предопределенная исключительная ситуация `STORAGE_ERROR` (Нехватка памяти). Возбуждение исключительных ситуаций будет рассмотрено в гл. 11.

**Пример.** Следующая процедура иллюстрирует только что введенные понятия

```

procedure EX is
  I : INTEGER := 0;
  J : INTEGER := K + I;
  -- Это об'явление неправильно, так как об'явле-
  -- ние величины K должно быть уже обработано
  -- к моменту начала обработки об'явления пере-
  -- менной J.
  K : INTEGER := 5;
  L : SMALL_RANGE;
  -- Это об'явление неправильно, так как обработка
  -- об'явления переменной L должна предшествовать
  -- обработке об'явления типа SMALL_RANGE.
  type SMALL_RANGE is range 1 .. 5;
  procedure IN1 is
    I : STRING ( 1 .. 5 );
  begin
    null;
    -- Здесь требуется наличие по крайней мере
    -- одного оператора.
    --          Область действия иденти-
    -- фикатора I, обозначающего об'ект строко-
    -- вого типа, начинается в точке об'явления
    -- этого идентификатора и заканчивается в
    -- строке end IN1;
    -- Идентификатор I, обозначающий об'ект це-
    -- лого типа, не видим в процедура IN1, не-
    -- смотря на то что процедура IN1 находит-
    -- ся в пределах его области действия.
    -- Эта ситуация служит примером перекрытия
    -- переменной I (пример, контрастирующий
    -- с перекрытием подпрограмм см. гл 5).
    -- Если потребуется обратиться к перемен-
    -- ной I целого типа в пределах тела IN1,
    -- то можно использовать составную форму
    -- записи: EX.I
  end IN1;
begin
  null;
end EX;
```

Вспомните, что некоторые ресурсы, такие, как идентификаторы циклов и блоков, объявляются неявно в конце декларативной части наиболее близкого охватывающего их блока или подпрограммы. Поэтому предполагается, что обработка этих неявных объявлений осуществляется после обработки явных объявлений.

### 6.1.2. Перекрывтие переменных и значений, использование квалификаторов

В процедуре EX имеется пример перекрывтия переменной. Могут перекрываться также и значения. В разд. 4.2.2 был приведен пример перекрывтия агрегатов. А вот пример перекрывтия перечисляемых литералов:

```
type FEM_NAMES is (MARY, CHARLIE, LAURIE, NANCY);
type MAL_NAMES is (JOHN, LAURIE, BOB, CHARLIE);
```

Здесь значение CHARLIE, являющееся перечисляемым литералом, перекрыто, так как оно обозначает два разных объекта, принадлежащие к двум различным типам. Такое перекрывтие не будет ошибкой, однако следует быть внимательным, чтобы обратиться к должному значению CHARLIE. Это достигается при помощи *квалификатора типа* (type qualifier). Например, при обращении к значению CHARLIE, принадлежащему к типу FEM\_NAMES, следует использовать квалификатор FEM\_NAMES' (CHARLIE), а при необходимости обратиться к значению CHARLIE, относящемуся к типу MAL\_NAMES, нужно воспользоваться квалификатором MAL\_NAMES' (CHARLIE). В общем случае *выражение с квалификатором* (qualified expression) имеет вид

тип\_или\_подтип' (выражение)

Для агрегатов форма записи выражения с квалификатором такова:

тип\_или\_подтип' агрегат

Необходимость в использовании квалификаторов возникает при употреблении выражений или агрегатов, тип которых нельзя однозначно определить по контексту. Например, в выражении CHARLIE < JOHN нет неоднозначности, поскольку JOHN принадлежит к типу MAL\_NAMES. Значение выражения – FALSE. Однако выражение CHARLIE < LAURIE некорректно, так как нельзя сделать однозначного заключения о том, к какому из двух типов принадлежат входящие в него перечисляемые литералы. Это выражение следует записать так:

MAL\_NAMES' (CHARLIE) < MAL\_NAMES' (LAURIE)

Если выбрана другая альтернатива, то нужно написать:

FEM\_NAMES' (CHARLIE) < FEM\_NAMES' (LAURIE)

## 6.2. ПРЕОБРАЗОВАНИЯ ТИПОВ

В первых трех главах книги были рассмотрены все типы Ады, за исключением приватных. Приватные типы будут описаны в гл. 7. В гл. 2 были кратко изложены правила преобразования для числовых типов. В языке Ада действуют строгие правила, запрещающие смешивание разных типов. Однако при использовании универсальных типов, к которым принадлежат универсальные целые и универсальные действительные числа, допускается некоторая свобода. Имеется в виду то, что в выражениях можно смешивать универсальные действительные и универсальные целые числа без каких-либо явных преобразований их типов.

**Пример.** Предположим, что имеются следующие объявления:

I : constant := 5;

Константа I относится к универсальному целому типу, она инициализируется значением универсального целого литерала 5

X : constant := 2.71;

Константа X принадлежит к универсальному действительному типу, она инициализируется значением универсального целого литерала 2.71

Тогда будут правильными такие выражения:

Выражение	Результат
5 * I	Получается значение (25) универсального целого типа; преобразования типа не требуется
3.0 * X	Получается значение (8.13) универсального действительного типа; преобразования типа не требуется
5.0 * I	Получается значение (25.0) универсального действительного типа; преобразования типа не требуется
I * X	Получается значение (13.55) универсального действительного типа; преобразования типа не требуется

В общем случае явные преобразования типа разрешены только для числовых, производных и регулярных типов. Общая форма *преобразования типа* такова:

тип\_или\_подтип (выражение)

При преобразовании типа может возникнуть исключительная ситуация CONSTRAINT\_ERROR (Нарушение\_уточнения), если значение вычисленного выражения не будет удовлетворять каким-либо из уточнений, наложенных на тип или подтип, к которому должен быть преобразован результат.

Для числовых типов (целых и действительных) выражение, следующее за именем типа\_или\_подтипа, может принадлежать к любому числовому типу. Значение этого выражения преобразуется к родительскому типу для указанного типа\_или\_подтипа. Член тип\_или\_подтип называется обозначением типа (type mark). Преобразование действительных значений в целые включает округление.

Преобразования для регулярных типов разрешены при условии, что типы индексов для каждого измерения обозначения типа и для результата вычисления выражения одинаковы или получены друг из друга, и типы согласующихся компонент также одинаковы или получены друг из друга.

**Пример.** Пусть имеются объявления:

```
type HOURS_WORKED is array (NATURAL range <>)
                           of NATURAL;
subtype HRS_WRKD_MONTHLY is HOURS_WORKED(1 .. 31);
subtype HRS_WRKD_WEEKLY is HOURS_WORKED(1 .. 7);
TOTAL_HOURS : HOURS_WORKED;
MONTHLY_DATA : HRS_WRKD_MONTHLY;
WEEKLY_DATA : HRS_WRKD_WEEKLY;
```

Тогда можно записать такие преобразования массивов:

HRS\_WRKD\_MONTHLY (TOTAL\_HOURS (44 .. 74))

Здесь границы 44 .. 74 массива TOTAL\_HOURS преобразуются в границы 1 .. 31 переменной регулярного типа, принадлежащей к типу HRS\_WRKD\_MONTHLY.

HOURS\_WORKED (MONTHLY\_DATA)

Здесь границы 1 .. 31 переменной MONTHLY\_DATA регулярного типа обеспечивают значения границ 1 .. 31 переменной, принадлежащей к регулярному типу HOURS\_WORKED.

HRS\_WRKD\_WEEKLY (MONTHLY\_DATA (13 .. 19))

Здесь границы 13 .. 19 вырезки из массива MONTHLY\_DATA становятся границами 1 .. 7 массива, тип которого — HRS\_WRKD\_WEEKLY.

Разрешены преобразования производных типов при условии, что и тип, указанный в обозначении типа, и тип значения выражения могут быть выведены друг из друга или имеют общий родительский тип.

**Пример.** Рассмотрим некоторые примеры возможных преобразований, затрагивающих производные типы. Пусть имеются такие объявления:

```
type DAYS is INTEGER range 1..130_000;
type JULIAN_DAYS is new DAYS range 1..366;
type MONTHLY_DAYS is new DAYS range 1..31;
CURR_JUL : JULIAN_DAYS;
LONG_DAYS : DAYS;
CURR_MONTH: MONTHLY_DAYS;
```

Тогда можно записать следующие преобразования:

```
LONG_DAYS := 5;
CURR_MONTH := MONTHLY_DAYS ( LONG_DAYS );
CURR_JUL := JULIAN_DAYS ( CURR_MONTH );
```

Дополнительные примеры будут приведены в разд. 6.4.

Если разрешено преобразование от одного типа к другому, то разрешено и обратное преобразование. При вызове подпрограмм обратное преобразование выполняется, когда в качестве фактического параметра с видом связи in out или out задается выражение, выполняющее преобразование типа переменной.

**Пример.** В данном примере используются массивы, объявленные в предыдущем примере. В дополнение к ним есть такое объявление процедуры:

```
procedure ADD_HRS (FORM_TOT_HRS : in out HOURS_WORKED; I : INTEGER);
```

Эту процедуру можно вызвать при помощи оператора

```
ADD_HRS (HOURS_WORKED (WEEKLY_DATA), II);
```

где II относится к целому типу. Здесь значение фактического параметра WEEKLY\_DATA преобразуется к значению типа HOURS\_WORKED, которое согласовывается с формальным параметром FORM\_TOT\_HRS. После завершения выполнения процедуры ADD\_HRS значение, возвращаемое параметром FORM\_TOT\_HRS (типа HOURS\_WORKED), преобразуется в значение типа HRS\_WRKD\_WEEKLY и передается переменной WEEKLY\_DATA. Тем самым производится неявное обратное преобразование к типу HRS\_WRKD\_WEEKLY:

```
HRS_WRKD_WEEKLY (FORM_TOT_HRS (1 .. 7))
```

которое осуществляется после завершения процедуры.

## 6.3. ИНСТРУКЦИИ ТРАНСЛЯТОРУ

*Инструкции транслятору* (pragmas) в языке Ада — это директивы, не носящие обязательного характера. Они, скорее, служат рекомендациями, которые транслятор может выполнить, а может и проигнорировать. Инструкции транслятору имеют следующий вид:

**pragma** идентификатор;

Если имеются аргументы, то форма инструкций становится такой:

`pragma` идентификатор (один или более аргументов, разделенных запятыми);

Существуют инструкции транслятору, определенные в языке, и инструкции, предлагаемые в конкретной реализации. Мы будем использовать только инструкции, определенные в языке. Пример инструкции транслятору, не имеющей аргумента:

`pragma PAGE;`

Инструкция `pragma PAGE` дает указание транслятору печатать следующую строку текста программы с новой страницы. Вот еще пример:

`pragma LIST (OFF);`

(другой возможный аргумент — `ON`). Эта инструкция дает указание транслятору отключить печать листинга программы. Эта инструкция будет действовать до тех пор, пока не встретится директива

`pragma LIST (ON);`

Инструкции `pragma LIST` и `pragma PAGE` можно употреблять в любом месте программы. Однако другие инструкции разрешается использовать только в определенных местах программы на языке Ада.

Как отмечалось в предыдущей главе, каждый вызов подпрограммы требует значительных дополнительных затрат времени. Для устранения этих накладных расходов, вызванных обработкой декларативной части и связыванием фактических и формальных параметров, можно воспользоваться инструкцией `pragma INLINE`. Эффект использования данной инструкции состоит в том, что в местах вызова подпрограммы будет непосредственно вставлен объектный код тела этой подпрограммы<sup>1)</sup>. Все остальные преимущества, обусловленные употреблением подпрограмм, сохраняются. Данная инструкция транслятору помещается в декларативной части после объявлений имен подпрограмм. Например:

`pragma INLINE (xx, yy, zz);`

Здесь `xx`, `yy` и `zz` — имена подпрограмм.

Инструкция `pragma ELABORATE` (Обработать) связана с трансляцией сегментов компиляции. Материал о сегментах компиляции представлен в гл. 9. Существуют также инструкции транслятору, связанные с механизмом задач языка Ада, например инструкция `pragma PRIORITY` (Приоритет). Эти инструкции будут рассмотрены в гл. 10. Существует инструкция `pragma INTERFACE` (Интерфейс), в которой указываются (в качестве аргументов) название другого языка программирования (например, COBOL) и имя подпрограммы (в данном примере на Коболе)<sup>2)</sup>.

Полный список инструкций транслятору, определенных в языке Ада, дан в приложении Б. Вот некоторые примеры:

`pragma SUPPRESS (RANGE_CHECK, ON => INDEX);`

Здесь дается разрешение транслятору отметить проверку того, что значение индекса массива находится в требуемом диапазоне.

`pragma PACK (ARRAY_NAME);`

`pragma OPTIMIZE (TIME);`

`pragma OPTIMIZE (SPACE);`

<sup>1)</sup> Получается так называемая открытая подпрограмма. — *Прим. перев.*

<sup>2)</sup> Эта инструкция позволяет использовать программу, написанную на другом языке программирования. — *Прим. перев.*

Две последние инструкции дают указание транслятору оптимизировать объектный код программы с целью максимального ускорения выполнения программы (OPTIMIZE (TIME)-Оптимизировать (время)) или с целью эффективного использования оперативной памяти (OPTIMIZE (SPACE)-Оптимизировать память)). Среди инструкций транслятору, определенных в языке Ада, отсутствует инструкция для условной компиляции. Однако можно ожидать, что такая инструкция окажется в списке инструкций, предлагаемых в конкретной реализации этого языка.

## 6.4. ОБЗОР ОСНОВНЫХ ОСОБЕННОСТЕЙ АДЫ

В данном разделе представлена программа, в которой используются подпрограммы, правила преобразования типов, инструкции транслятору, структуры с вариантами, ссылочные типы, производные типы и все виды операторов языка Ада, с которыми к настоящему моменту познакомился читатель. Эта программа иллюстрирует все основные особенности языка, описанные в первых шести главах настоящей книги.

Программа вычисляет проценты по двум видам финансовых документов. В целях упрощения здесь рассматриваются только ценные бумаги со скидкой, например векселя министерства финансов США, и ценные бумаги, по которым платятся проценты при их выкупе, например сертификаты или коммерческие бумаги.

Доход по тем бумагам, по которым платятся проценты при выкупе, вычисляется в соответствии с их ценой по формуле

$$YLD = ( 1 + DAYS\_ISS\_MAT / D\_IN\_YEAR * S\_RATE - \\ ( QT\_PRICE + DAYS\_ISS\_SET / D\_IN\_YEAR * \\ S\_RATE ) ) / ( QT\_PRICE + DAYS\_ISS\_SET / \\ D\_IN\_YEAR * S\_RATE ) * D\_IN\_YEAR / \\ DAYS\_SET\_MAT;$$

Переменные в этой формуле обозначают следующее:

Переменная	Смысл
DAYS_ISS_SET	Количество дней от даты выпуска ценной бумаги до даты ее приобретения
D_IN_YEAR	Количество дней в году
DAYS_ISS_MAT	Количество дней от даты выпуска ценной бумаги до даты ее выкупа
DAYS_SET_MAT	Количество дней от даты приобретения до даты выкупа
QT_PRICE	Курсовая стоимость ценной бумаги, деленная на 100
S_RATE	Годовой процент или процент, выплачиваемый по купону (десятичное число)
YLD	Годовой доход по ценной бумаге (десятичное число), которая хранится у вкладчика до даты выкупа или до даты погашения (если она сохранила стоимость к этой дате)

Вычисление процента по скидке для цены со скидкой производится по формуле:

$$DISC\_RATE = (100 - DISC\_PRICE) / 100 * D\_IN\_YEAR / DAYS\_SET\_MAT$$

где DISC\_RATE – процент по скидке (десятичное число), а DISC\_PRICE – цена со скидкой.

Для сравнения выплачиваемых процентов (т. е. YLD, которые являются доходом на денежном рынке) с процентами по скидке используется формула:

$$YLD\_OF\_DISC = DISC\_RATE * 100 / DISC\_PRICE$$

Предполагается, что ни дата приобретения, ни дата выкупа, ни дата выпуска ценной бумаги не должны попадать на праздничные или выходные дни. В упр. 1 предлагается ввести дополнительную проверку этого условия.

В каждой входной строке данных располагаются сведения об одной ценной бумаге в следующем формате:

Позиции	Данные
1-15	Вид ценной бумаги AT_MATURITY (выплата дохода при выкупе бумаги) или DISCOUNT (бумага со скидкой)
16-25	Наименование ценной бумаги
26-31	Дата приобретения в формате ГГММДД
32-37	Дата выкупа
38-48	Способ подсчета дней (ACT_360 и т. д. - см. далее текст программы)

В зависимости от вида ценной бумаги далее идут различные поля. Для бумаг вида AT\_MATURITY:

Позиции	Данные
49-54	Дата выпуска
55-60	Установленный процент
61-68	Курсовая стоимость (два знака после десятичной точки)

Для бумаг вида DISCOUNT:

49-57	Цена со скидкой (два знака после десятичной точки)
-------	---

Признаком конца данных служит входная строка со значением «5555588888» в поле названия ценной бумаги.

Для каждой ценной бумаги программа вычисляет доход по ней. Для бумаг со скидкой вначале вычисляется процент по скидке. Бумаги помещаются в списке в порядке возрастания дат выкупа.

Размер дохода как функцию от даты выкупа иногда называют кривой дохода. После обработки всех ценных бумаг следует проверить кривую дохода на выпуклость (нормальный характер зависимости). Тест на выпуклость заключается в следующем. Для каждой ценной бумаги доход должен равняться по крайней мере среднему значению доходов по предшествующей и последующей ценным бумагам.

Последним допущением будет то, что даты приобретения должны находиться близко друг к другу. Это упрощает задачу. В упр. 2 даны более реалистические условия.

### Программа YIELD\_COMPUTATION

```
with TEXT_IO; use TEXT_IO;
procedure YIELD_COMPUTATION is
  type INTEREST is digits 13;
  type DISCOUNT_INTEREST is new INTEREST;
  type AT_MAT_INTEREST is new INTEREST;
  -- Не следует путать проценты по скидке и
  -- проценты по другим видам ценных бумаг.
  -- Они обозначают разные вещи. Здесь впол-
  -- не уместным будет использование произ-
  -- водных типов.
  WORK_AT_MAT_INT_1, WORK_AT_MAT_INT_2 :
    AT_MAT_INTEREST;
  -- Переменные, названия которых начинаются
  -- с WORK, - это рабочие переменные.
```

```

type INTEREST_KIND is
  (COUPON, AT_Maturity, DISCOUNT);
  -- Здесь не рассматриваются проценты, вы-
  -- плачиваемые по купонам. Перечисляемое
  -- значение COUPON приведено только для
  -- полноты списка.
package INT_KIND_IO is new ENUMERATION_IO
  (INTEREST_KIND);
use INT_KIND_IO;
CURR_INT_KIND : INTEREST_KIND;
CURR_SEC_NAME : STRING(1 .. 10);
type DAY_COUNT_BASIS is
  (ACT_ACT, ACT_360, M_30_Y_360);
  -- Здесь приняты такие условные обозначения:
  -- ACT_ACT означает, что при вычислениях
  -- учитывается действительное количество дней
  -- месяце (28-31) и в году (365-366);
  -- ACT_360 означает, что при вычислениях учи-
  -- тывается действительное количество дней в
  -- месяце, но считается, что в году 360 дней;
  -- M_30_Y_360 принимается, что в месяце -
  -- 30 дней, а в году - 360 дней.
package DAY_KIND_IO is new ENUMERATION_IO
  (DAY_COUNT_BASIS);
use DAY_KIND_IO;
type PRICES is digits 13;
type DISCOUNT_PRICES is new PRICES;
  -- Опять-таки цены со скидкой означают иное,
  -- чем цены других видов ценных бумаг.
type DAY is
  (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
   FRIDAY, SATURDAY, SUNDAY);
subtype WEEKEND is DAY range SATURDAY..SUNDAY;
type DAY_INT is range 1 .. 31;
type JULIAN_DAYS is range 1 .. 366;
type MONTH is
  (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
   JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER,
   DECEMBER);
type YEAR is range 0 .. 2050;
type MONTH_INT is range 1 .. 12;
package DAY_IO is new ENUMERATION_IO(DAY);
use DAY_IO;
package MONTH_IO is new ENUMERATION_IO(MONTH);
use MONTH_IO;
package YEAR_IO is new INTEGER_IO(YEAR);
use YEAR_IO;
package MONTH_INT is new INTEGER_IO(MONTH_INT);
use MONTH_INT;
package DAY_INT_IO is new INTEGER_IO(DAY_INT);
use DAY_INT_IO;
type DATE is
  record
    WEEK_DAY : DAY;
    MONTH_NAME : MONTH;
    MONTH_NO : MONTH_INT;
    DAY_NO : DAY_INT;
  end record;

```



```

YTD_DAYS : JULIAN_DAYS;
TOTAL_DAYS : INTEGER;
YEAR_NO : YEAR;
end record;
BASE_DATE : constant DATE :=
(MONDAY, JANUARY, 1, 1, 1, 1, 1984);
-- В данной программе предполагается, что во
-- входных данных указываются только даты,
-- следующие после 1 января 1984г.
BASE_LEAP : constant INTEGER :=
INTEGER(BASE_DATE.YEAR_NO) / 4 +
INTEGER(BASE_DATE.YEAR_NO) / 400 -
INTEGER(BASE_DATE.YEAR_NO) / 100;
-- Эта константа равна количеству високосных
-- лет в период от 0-го года до года
-- BASE_DATE.YEAR_NO.
type DAYS_IN_MONTH is array(MONTH_INT, BOOLEAN)
of DAY_INT;
ACTUAL_DAYS_IN_YEAR : constant DAYS_IN_MONTH :=
( (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),
  (31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31) );
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
D_IN_YEAR : NATURAL := 360;
DAYS_ISS_SET : NATURAL;
DAYS_ISS_MAT : NATURAL;
DAYS_SET_MAT : NATURAL;
type SECURITY(INT_PAYM : INTEREST :=
DISCOUNT) is
record
  SEC_NAME : STRING(1 .. 10);
  SETL_DATE : DATE;
  MAT_DATE : DATE;
  DAY_KIND : DAY_COUNT_BASIS;
  YLD : INTEREST;
  case INT_PAYM is
    when AT_MATURITY =>
      ISSUE_DATE : DATE;
      S_RATE : AT_MAT_INTEREST;
      QT_PRICE : PRICES;
    when DISCOUNT =>
      DISC_PRICE : DISCOUNT_PRICES;
      DISC_RATE : DISCOUNT_INTEREST;
    when others => null;
  end case;
end record;
CURR_SECURITY_1 : SECURITY :=
(AT_MATURITY, "1234567890", BASE_DATE,
BASE_DATE, M_30_Y_360, 10.0, BASE_DATE,
100.0, 100.0 );
-- Эта структура инициализируется при помощи
-- позиционного агрегата. Значение дискри-
-- минанта равно AT_MATURITY.
CURR_SECURITY_2 : SECURITY :=
(INT_PAYM => DISCOUNT,
SEC_NAME => "1234512345",
SETL_DATE => BASE_DATE,
MAT_DATE => BASE_DATE,

```

```

DAY_KIND    => M_30_Y_360,
YLD         => 10.0,
DISC_PRICE  => 99.0,
DISC_RATE   => 10.0 );
-- Эта структура инициализируется при помощи
-- агрегата с поименованными компонентами.
-- Значение дискриминанта равно DISCOUNT.
package INTEREST_IO is new FLOAT_IO(INTEREST);
use INTEREST_IO;
package PRICES_IO is new FLOAT_IO(PRICES);
use PRICES_IO;
package DISC_IO is new FLOAT_IO(DISCOUNT_INTEREST);
use DISC_IO;
package AT_MAT_IO is new FLOAT_IO(AT_MAT_INTEREST);
use AT_MAT_IO;
LEAP_YEAR : BOOLEAN;
INPUT_LEAP : INTEGER;
GOOD_MAT_DATE : BOOLEAN;
GOOD_SETL_DATE : BOOLEAN;
GOOD_ISSUE_DATE : BOOLEAN;
BAD_LINE : BOOLEAN;
-- При обнаружении ошибки во входных данных
-- переменная BAD_LINE получает значение
-- TRUE.
type SEC_NODE;
type SEC_PTR is access SEC_NODE;
type SEC_NODE is
  record
    SEC_INFO : SECURITY;
    SEC_NEXT : SEC_PTR;
  end record;
CURR_PTR, TOP_PTR, ANY_PTR, PREV_PTR : SEC_PTR;
NORM_CURVE : BOOLEAN;
-- Эта переменная устанавливается равной TRUE,
-- если кривая дохода имеет нормальный вид.
pragma PAGE;
-- Устанавливается печать листинга программы
-- с новой строки.
function IS_VALID_DATE (FORM_DATE : DATE)
  return BOOLEAN is
  -- Эта функция дает значение TRUE, если
  -- дата, представленная переменными
  -- FORM_DATE.MONTH_NO, FORM_DATE.DAY_NO
  -- и FORM_DATE.YEAR_NO, правильна. В про-
  -- тивном случае вырабатывается значение
  -- FALSE.
begin
  pragma LIST(OFF);
  -- Последовательность операторов этой
  -- подпрограммы не нужно печатать.
  -- Эта последовательность идентична тексту
  -- одноименной подпрограммы из программы
  -- ACCR_INTEREST (см. гл.5). Этот текст
  -- нужно сюда вставить.
  pragma LIST(ON);
  -- Возобновить печать текста программы.
end IS_VALID_DATE;

```

```

procedure FILL_IN_DATE
  (PROC_F_DATE : in out DATE;
   GOOD_DATE   : out BOOLEAN) is
  -- Эта процедура вычисляет значения оставших-
  -- ся компонент структуры PROC_F_DATE, причем
  -- предполагается, что значения известных
  -- компонент - FORM_DATE.MONTH_NO,
  -- FORM_DATE.DAY_NO, FORM_DATE.YEAR_NO обра-
  -- зуют правильную дату. Если это не так, то
  -- переменная GOOD_DATE получает значение
  -- FALSE.
  -- В формальной части данной процедуры имеются
  -- два формальных параметра: PROC_F_DATE ком-
  -- бинированного типа и GOOD_DATE логического
  -- типа.
begin
  pragma LIST(OFF);
  -- Последовательность операторов для данной
  -- подпрограммы не следует печатать. Эта по-
  -- следовательность идентична тексту одно-
  -- именной подпрограммы из программы
  -- ACCR_INTEREST (см. гл.5). Этот текст сле-
  -- дует сюда вставить.
  pragma LIST(ON);
  -- Возобновить печать листинга программы.
end FILL_IN_DATE;
function "-" (X, Y : DATE)
  -- Эта функция подсчитывает количество прошед-
  -- ших дней в предположении, что в месяце на-
  -- считывается 30 дней, а в году - 360
  -- (M_30_Y_360). Это пример использования пере-
  -- крытой операции "-".
  return NATURAL is
begin
  if X.TOTAL_DAYS < Y.TOTAL_DAYS
  then
    return 0;
  else
    return 360 * INTEGER(X.YEAR_NO-Y.YEAR_NO) +
      30 * INTEGER(X.MONTH_NO-Y.MONTH_NO) +
      INTEGER(X.DAY_NO-Y.DAY_NO);
  end if;
end "-";
pragma PAGE;
-- Начать печать операторов исполняемой части
-- программы с новой страницы.
begin
  GET(CURR_INT_KIND);
  GET(CURR_SEC_NAME);
  while CURR_SEC_NAME /= "5555588888"
  loop
    case CURR_INT_KIND is
      when AT_MATURITY =>
        CURR_SECURITY_1.SEC_NAME := CURR_SEC_NAME;
        -- Считать со входной строки оставшиеся
        -- данные.
        GET(CURR_SECURITY_1.SETL_DATE.YEAR_NO,2);
        GET(CURR_SECURITY_1.SETL_DATE.MONTH_NO,2);
    end case;
  end loop;
end;

```

```

GET(CURR_SECURITY_1.SETL_DATE.DAY_NO,2);
GET(CURR_SECURITY_1.MAT_DATE.YEAR_NO,2);
GET(CURR_SECURITY_1.MAT_DATE.MONTH_NO,2);
GET(CURR_SECURITY_1.MAT_DATE.DAY_NO,2);
GET(CURR_SECURITY_1.DAY_KIND);
GET(CURR_SECURITY_1.ISSUE_DATE.YEAR_NO,2);
GET(CURR_SECURITY_1.ISSUE_DATE.MONTH_NO,2);
GET(CURR_SECURITY_1.ISSUE_DATE.DAY_NO,2);
GET(CURR_SECURITY_1.S_RATE,6);
GET(CURR_SECURITY_1.OT_PRICE,8);
-- Проверим правильность каждой даты и
-- заппомним остальные компоненты путем
-- обращения к процедуре FILL_IN_DATE.
FILL_IN_DATE(CURR_SECURITY_1.SETL_DATE,
              GOOD_SETL_DATE);
FILL_IN_DATE(CURR_SECURITY_1.MAT_DATE,
              GOOD_MAT_DATE);
FILL_IN_DATE(CURR_SECURITY_1.ISSUE_DATE,
              GOOD_ISSUE_DATE);
if not (GOOD_SETL_DATE and
        GOOD_MAT_DATE and
        GOOD_ISSUE_DATE and
        CURR_SECURITY_1.MAT_DATE.TOTAL_DAYS >
        CURR_SECURITY_1.SETL_DATE.TOTAL_DAYS
        and
        CURR_SECURITY_1.SETL_DATE.TOTAL_DAYS >
        CURR_SECURITY_1.ISSUE_DATE.TOTAL_DAYS)
then
    BAD_LINE := TRUE;
    PUT(" Bad line ");
    PUT( CURR_SEC_NAME );
    NEW_LINE;
end if;
if not BAD_LINE
then
    -- Вычислить доход с учетом выбран-
    -- ного способа подсчета дней.
    case CURR_SECURITY_1.DAY_KIND is
    when ACT_360 =>
        DAYS_SET_MAT :=
            CURR_SECURITY_1.MAT_DATE.TOTAL_DAYS
        - CURR_SECURITY_1.SETL_DATE.TOTAL_DAYS;
        DAYS_ISS_SET :=
            CURR_SECURITY_1.SETL_DATE.TOTAL_DAYS
        - CURR_SECURITY_1.ISSUE_DATE.TOTAL_DAYS;
    when M_30_Y_360 =>
        DAYS_SET_MAT :=
            CURR_SECURITY_1.MAT_DATE
        - CURR_SECURITY_1.SETL_DATE;
        DAYS_ISS_SET :=
            CURR_SECURITY_1.SETL_DATE
        - CURR_SECURITY_1.ISSUE_DATE;
    -- Операция "-" перекрыта.
    when others =>
        PUT(" What security is this ");
        PUT( CURR_SEC_NAME );
        NEW_LINE;
    end case;
end if;

```

```

end case;
-- Здесь необходимо довольно большое
-- количество преобразований значе-
-- ний к типу AT_MAT_INTEREST.
WORK_AT_MAT_INT_1 :=
  CURR_SECURITY_1.S_RATE /
  AT_MAT_INTEREST(D_IN_YEAR);
WORK_AT_MAT_INT_2 :=
  AT_MAT_INTEREST(CURR_SECURITY_1.QT_PRICE)
  + AT_MAT_INTEREST(DAYS_ISS_SET) /
  WORK_AT_MAT_INT_1;
CURR_SECURITY_1.YLD := INTEREST(
  (1.0 + AT_MAT_INTEREST(DAYS_ISS_MAT)/
  WORK_AT_MAT_INT_1 -
  WORK_AT_MAT_INT_2) /
  WORK_AT_MAT_INT_2 *
  AT_MAT_INTEREST(D_IN_YEAR) /
  AT_MAT_INTEREST(DAYS_SET_MAT) );
-- Создать элемент, который будет
-- занесен в список.
CURR_PTR := new SEC_NODE;
CURR_PTR.SEC_INFO := CURR_SECURITY_1;
end if;
when DISCOUNT =>
  CURR_SECURITY_2.SEC_NAME :=
    CURR_SEC_NAME;
  -- Считать со входной строки оставшиеся
  -- данные.
  GET(CURR_SECURITY_2.SETL_DATE.YEAR_NO,2);
  GET(CURR_SECURITY_2.SETL_DATE.MONTH_NO,2);
  GET(CURR_SECURITY_2.SETL_DATE.DAY_NO,2);
  GET(CURR_SECURITY_2.MAT_DATE.YEAR_NO,2);
  GET(CURR_SECURITY_2.MAT_DATE.MONTH_NO,2);
  GET(CURR_SECURITY_2.MAT_DATE.DAY_NO,2);
  GET(CURR_SECURITY_2.DAY_KIND);
  GET(CURR_SECURITY_2.DISC_PRICE,8);
  -- Проверить правильность дат и запомнить
  -- значения других компонент при помощи
  -- обращения к процедуре FILL_IN_DATE.
  FILL_IN_DATE(CURR_SECURITY_1.SETL_DATE,
    GOOD_SETL_DATE);
  FILL_IN_DATE(CURR_SECURITY_1.MAT_DATE,
    GOOD_MAT_DATE);
  if not (GOOD_SETL_DATE and
    GOOD_MAT_DATE and
    CURR_SECURITY_2.MAT_DATE.TOTAL_DAYS >
    CURR_SECURITY_2.SETL_DATE.TOTAL_DAYS)
  then
    BAD_LINE := TRUE;
    PUT(" Bad line ");
    PUT( CURR_SEC_NAME );
    NEW_LINE;
  end if;
  if not BAD_LINE
  then
    -- Вычислить доход с учетом применяе-
    -- мого способа подсчета дней.

```

```

case CURR_SECURITY_2.DAY_KIND is
  when ACT_360 =>
    DAYS_SET_MAT :=
      CURR_SECURITY_2.MAT_DATE.TOTAL_DAYS
    - CURR_SECURITY_2.SETL_DATE.TOTAL_DAYS;
    when M_30_Y_360 =>
      DAYS_SET_MAT :=
        CURR_SECURITY_1.MAT_DATE -
        CURR_SECURITY_1.SETL_DATE;
      -- Операция "-" перекрыта.
    when others =>
      PUT(" What security is ");
      PUT( CURR_SEC_NAME );
      NEW_LINE;
end case;
-- Здесь опять-таки требуются пре-
-- образования значений к типам
-- DISCOUNT_INTEREST и
-- AT_MAT_INTEREST.
CURR_SECURITY_2.DISC_RATE :=
  (100.0 - DISCOUNT_INTEREST(
    CURR_SECURITY_2.DISC_PRICE) /
  100.0 * DISCOUNT_INTEREST(
    D_IN_YEAR) / DISCOUNT_INTEREST
  (DAYS_SET_MAT));
CURR_SECURITY_2.YLD :=
  AT_MAT_INTEREST(
    CURR_SECURITY_2.DISC_RATE)*100.0 /
  AT_MAT_INTEREST(
    CURR_SECURITY_2.DISC_RATE);
-- Создадим элемент, который будет
-- помещен в список.
CURR_PTR := new SEC_NODE;
CURR_PTR.SEC_INFO :=
  CURR_SECURITY_2;
end if;
when COUPON => null;
end case;
-- Поместим правильные сведения о ценной
-- бумаге в список.
if not BAD_LINE
  then
    ANY_PTR := TOP_PTR;
    while
      ANY_PTR.SEC_INFO.MAT_DATE.TOTAL_DAYS <
      CURR_PTR.SEC_INFO.MAT_DATE.TOTAL_DAYS
    or
      ANY_PTR /= NULL
    loop
      PREV_PTR := ANY_PTR;
      ANY_PTR := ANY_PTR.SEC_NEXT;
    end loop;
    if ANY_PTR = TOP_PTR
      then
        CURR_PTR.SEC_NEXT := TOP_PTR;
        TOP_PTR := CURR_PTR;
      else

```

```

    PREV_PTR.SEC_NEXT := CURR_PTR;
    CURR_PTR.SEC_NEXT := ANY_PTR;
  end if;
  SKIP_LINE;
  GET(CURR_INT_KIND);
  GET(CURR_SEC_NAME);
  end if;
end loop;
-- Вид кривой - нормальный ?
ANY_PTR := TOP_PTR;
NORM_CURVE := TRUE;
while ANY_PTR /= NULL
loop
  CURR_PTR := ANY_PTR;
  ANY_PTR := ANY_PTR.SEC_NEXT;
  if ANY_PTR /= NULL
    and then
      ANY_PTR.SEC_NEXT /= NULL
    and then
      -- Здесь кривая проверяется на вы-
      -- пуклость.
      ANY_PTR.SEC_INFO.YLD <
      (CURR_PTR.SEC_INFO.YLD +
      ANY_PTR.SEC_NEXT.SEC_INFO.YLD)
      / 2.0
    then
      NORM_CURVE := FALSE;
      exit;
    end if;
  end loop;
  if NORM_CURVE
    then
      PUT(" Normal yield curve ");
    else
      PUT(" Maybe the curve is inverted ");
    end if;
end YIELD_COMPUTATION;

```

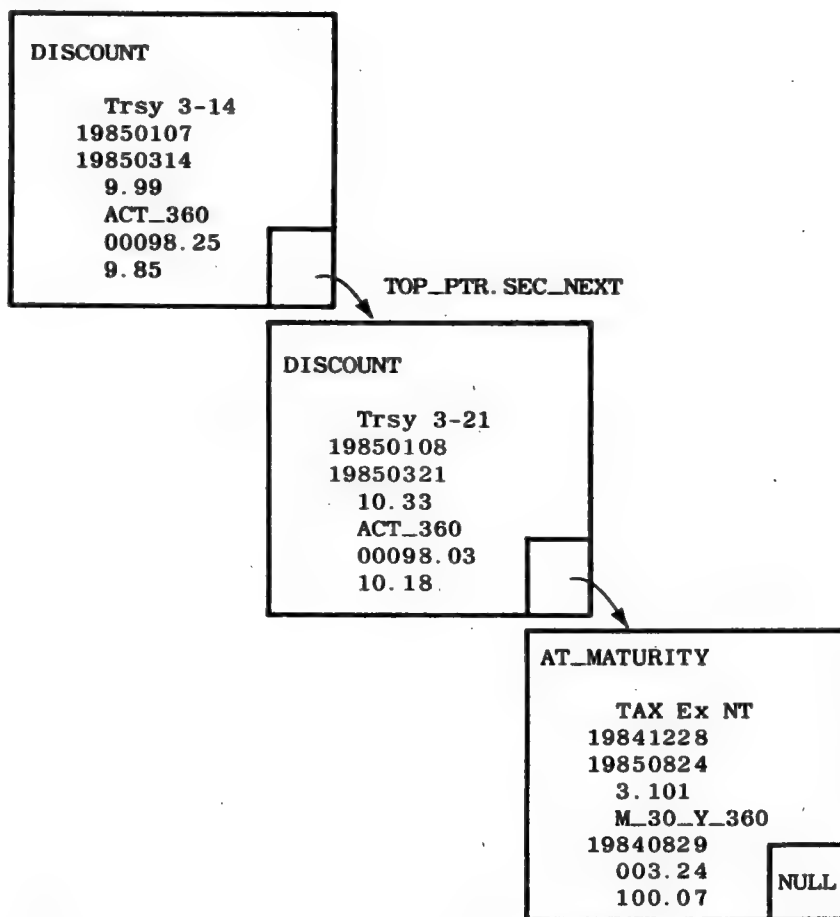
Пример входных данных и вид списка приведены на рис. 6.1.

## УПРАЖНЕНИЯ

1. Внесите изменения в программу YIELD\_COMPUTATION, которые обеспечат контроль того, чтобы дата приобретения ценной бумаги, дата ее выкупа и дата выпуска не попадали на выходные и праздничные дни.
2. Пусть входные строки данных для программы YIELD\_COMPUTATION отсортированы в порядке возрастания дат приобретения. Измените программу YIELD\_COMPUTATION так, чтобы проверка выпуклости кривой дохода выполнялась только для ценных бумаг, имеющих одинаковые даты приобретения.
3. Модифицируйте программу YIELD\_COMPUTATION с тем, чтобы строился список ценных бумаг такого вида, что добавление любой ценной бумаги в него нарушало бы его нормальность. (При удалении ценной бумаги из списка выпуклость кривой дохода сохраняется.) Заметьте, что бумаги, попавшие в этот список, видимо, стоит покупать.

Позиция 1	Позиция 15	Позиция 25	
DISCOUNT	Trsy 3-14	850107850314ACT_360	00098.25
DISCOUNT	Trsy 3-21	850108850321ACT_360	00098.03
AT_MATURITY	TAX Ex NT	850108850904M_30_Y_360	840829003.24100.07
DISCOUNT	5555588888		

TOP\_PTR



Примечание. Даты относятся к типу DATE и имеют намного больше компонент, чем показано на рисунке.

Рис. 6.1. Пример входных данных и значения списка для подпрограммы YIELD\_COMPUTATION.



## 7.1. СПЕЦИФИКАЦИИ ПАКЕТОВ И ПРИВАТНЫЕ ТИПЫ

В первых шести главах были освещены «традиционные» свойства языка программирования Ада. Большинство из этих свойств можно найти в эквивалентных формах у таких общепризнанных языков, как Фортран, ПЛ/1, Кобол, Бейсик и Паскаль.

Оставшиеся главы будут посвящены тем особенностям Ады, которые олицетворяют явный отход от этих популярных языков. Начнем с пакетов — одного из наиболее полезных инструментов для разработки программ.

### 7.1.1. Спецификации пакетов

*Пакеты* Ады наряду с подпрограммами и задачами (см. гл. 10) являются программными сегментами. Пакет в Аде представляет собой совокупность программных ресурсов, таких, как типы, объекты и подпрограммы. Однако эта совокупность ресурсов подчиняется строгим правилам. Эти правила регулируют то, какая информация делается доступной пользователю, и то, какая доля внутренних процессов будет ему показана. Разработчик пакета может применять эти правила достаточно гибко. Программист сам устанавливает желаемую границу между пакетом и его пользователями.

Пакет, как и процедура или функция, может иметь спецификацию и тело. Таким образом, пакет состоит из двух частей: спецификации и тела.

*Объявлением пакета* называется его спецификация, за которой ставится точка с запятой. Здесь наблюдается аналогия с объявлением подпрограммы. *Спецификация пакета* имеет форму

```
package имя_пакета is  
базисные_объявления_или_фразы_использования_или_фразы_представления  
end имя_пакета
```

Заметьте, что объявление пакета получится, если после члена «end имя\_пакета» поставить символ «;». Если в пакете имеется приватная часть, то спецификация пакета принимает вид

```
package имя_пакета is  
базисные_объявления_или_фразы_использования_или_фразы_представления  
private  
базисные_объявления_или_фразы_использования_или_фразы_представления  
end имя_пакета
```

Имя\_пакета после зарезервированного слова end можно не указывать.

**Пример.** Ниже приведена спецификация пакета с использованием некоторых объявлений из программы ACCR\_INTEREST (гл. 5).

```

package DAYS_MODULE is
type DAY is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
             FRIDAY, SATURDAY, SUNDAY);
subtype WEEKEND is DAY range SATURDAY .. SUNDAY;
type DAY_INT is range 1 .. 31;
type JULIAN_DAYS is range 1 .. 366;
type MONTH is (JANUARY, FEBRUARY, MARCH, APRIL, MAY,
               JUNE, JULY, AUGUST, SEPTEMBER,
               OCTOBER, NOVEMBER, DECEMBER);
type YEAR is range 0 .. 2050;
type DATE is
  record
    WEEK_DAY : DAY;
    MONTH_NAME : MONTH;
    MONTH_NO : MONTH_INT;
    DAY_NO : DAY_INT;
    YTD_DAYS : JULIAN_DAYS;
    TOTAL_DAYS : INTEGER;
    YEAR_NO : YEAR;
  end record;
BASE_DATE : constant DATE :=
  (MONDAY, JANUARY, 1, 1, 1, 1, 1984);
BASE_LEAP : constant INTEGER :=
  INTEGER(BASE_DATE.YEAR_NO) / 4 +
  INTEGER(BASE_DATE.YEAR_NO) / 400 -
  INTEGER(BASE_DATE.YEAR_NO) / 100;
type DAYS_IN_MONTH is array (MONTH_INT, BOOLEAN)
  of DAY_INT;
ACTUAL_DAYS_IN_YEAR : constant DAYS_IN_MONTH :=
  ( (31, 31), (28, 29), (31, 31), (30, 30),
    (31, 31), (30, 30), (31, 31), (31, 31),
    (30, 30), (31, 31), (30, 30), (31, 31) );
-- ПОМЕТКА для возможной фразы представления
end DAYS_MODULE
-- Наличие символа ";" в этом
-- месте превратило бы данную
-- спецификацию в об'явление
-- пакета.

```

В этой спецификации пакета отсутствует приватная часть и использованы лишь *базисные объявления*. Базисными объявлениями здесь являются объявления типов и связанных с ними объектов. В качестве базисных объявлений можно использовать любые из объявлений, введенных в предыдущих главах: объявления чисел, объектов, типов, подтипов и подпрограмм. Например, в спецификации пакета `CHECK_DATES`, приведенной ниже, встречаются примеры объявлений функций и подпрограмм, употребляемые в качестве базисных объявлений. Другими возможными видами базисных объявлений являются объявления задач (см. гл. 10), объявления исключительных ситуаций (см. гл. 11) и объявления других пакетов (как, например, в спецификации пакета `CHECK_DATES_ALT`, приведенной далее). Кроме того, разрешены объявления переименования и объявления отложенных констант. Подробные сведения об этих видах объявлений даются далее в настоящей главе.

Фразы представления также могут присутствовать в спецификации пакета. *Фразы представления* (representation clauses) содержат указания о том, в каком виде должны быть представлены типы в ЭВМ. Эти фразы зачастую используются для того, чтобы ограничить транслятор в свободе выбора одного из множества возможных вариантов внутреннего представления для требуемого типа. Например, вместо отмеченного комментария в спецификации пакета `DAYS_MODULE` можно поместить такую фразу

представления:

```
-- Фраза представления
for DAY use (MONDAY => 1, TUESDAY => 2,
             WEDNESDAY => 3, THURSDAY => 4,
             FRIDAY => 5, SATURDAY => 6,
             SUNDAY => 7 ) ;
```

Эта фраза представления дает указание транслятору использовать целые числа в диапазоне от 1 до 7 для внутреннего представления величин типа DAY. Без данной фразы транслятор мог бы выбрать для реализации этого типа иной набор целых числовых кодов, скажем, от 0 до 6. Фразы представления играют важную роль при обеспечении совместимости программ на Аде (например, с файлами, созданными посредством других языков программирования).

В заключение отметим, что спецификации пакетов могут также содержать *фразы использования* (use clauses). Общая форма фразы использования такова:

use имя\_пакета;

Здесь можно указать имена нескольких пакетов, отделенные друг от друга запятыми.

**Пример.** Ниже приведена спецификация пакета с применением объявлений подпрограмм из программы ACCR\_INTEREST, содержащей фразу использования use.

```
with DAYS_MODULE ;
-- фраза подключения контекста with делает видимым
-- ми об'явления ресурсов из спецификации пакета
-- DAYS_MODULE.
package CHECK_DATES is
-- Далее следует фраза использования use .
use DAYS_MODULE;
-- Эта фраза использования позволяет употреблять
-- идентификаторы из спецификации пакета
-- DAYS_MODULE без применения составных имен.
-- Таким образом, можно использовать простое
-- имя типа - DATE вместо употребления префиксной
-- формы записи - DAYS_MODULE.DATE. Идентификаторы
-- в этом случае становятся "непосредственно
-- видимыми" (см. разд.7.3).
function IS_VALID_DATE (FORM_DATE : DATE)
    return BOOLEAN;
-- Это об'явление функции является
-- базисным об'явлением.
procedure FILL_IN_DATE
    ( PROC_F_DATE : in out DATE;
      GOOD_DATE   : out BOOLEAN );
-- Это об'явление процедуры является
-- базисным об'явлением.
TODAYS_DATE : DATE;
end CHECK_DATES -- Символ ";" превратит данную
                -- спецификацию пакета в
                -- об'явление пакета.
```

Следует подчеркнуть, что в спецификацию пакета нельзя помещать тела подпрограмм и других пакетов. Например, недопустимо включать тело процедуры FILL\_IN\_DATE в спецификацию пакета CHECK\_DATES.

## 7.1.2. Видимость в спецификациях пакетов

В только что приведенной спецификации пакета отсутствовала приватная часть. Если же приватная часть пакета присутствует, то объявления, следующие за зарезервированным словом `private`, образуют так называемую приватную часть пакета. Объявления, размещенные до слова `private`, составляют видимую часть пакета. В *видимой части* перечисляются те ресурсы, которые могут быть сделаны известными и доступными пользователю. В *приватной же части* перечисляются те ресурсы, которые необходимы для должного функционирования пакета, но на которые не может сослаться пользователь, так как они ему недоступны. Пользователю ничего не известно об объявлениях в приватной части, даже если приватные типы упоминаются и используются в видимой части спецификации пакета. Это — так называемый принцип «черного ящика», т. е. системы, которой можно продуктивно пользоваться, но внутреннее устройство которой неизвестно.

Видимую часть пакета можно сделать доступной для других программных сегментов с помощью ряда средств. Типичный способ установления видимости для нужного пакета — это употребление фразы подключения контекста (`with clause`), предшествующей программному сегменту. Так, например, перед объявлением пакета `CHECK_DATES` может располагаться фраза подключения контекста

`with DAYS_MODULE;`

Фраза `with` экспортирует ресурсы пакета `DAYS_MODULE` для использования их в пакете `CHECK_DATES`. Термин «экспортирует» является предпочтительным термином языка Ада для обозначения того, что объявления видимой части пакета становятся доступными и известными. Предполагается, что пакет `DAYS_MODULE` транслируется отдельно и доступен также и другим программным сегментам. (Подробности см. в гл. 9.) Предоставляемые пакетом `DAYS_MODULE` ресурсы можно использовать в пакете `CHECK_DATES`, если воспользоваться составными именами. Например, внутри пакета `CHECK_DATES` можно употребить идентификаторы `DAYS_MODULE.BASE_DATE` или `DAYS_MODULE.BASE_LEAP`.

Фраза использования `use` в спецификации пакета `CHECK_DATES` обеспечивает дополнительные удобства для программиста, так как она упрощает применение видимых идентификаторов из пакета `DAYS_MODULE`. После обработки фразы `use` видимые идентификаторы из пакета `DAYS_MODULE` можно записывать без префикса `DAYS_MODULE`. Сложные правила, касающиеся видимости, области действия идентификаторов и употребления фразы `use`, будут более детально изложены в разд. 7.3.

Другим способом установления видимости пакета может служить размещение объявления этого пакета в спецификации использующего его пакета.

**Пример.** Можно воспользоваться таким альтернативным способом задания спецификации пакета `CHECK_DATES` (при этом будет обеспечен тот же самый род видимости, что и в предыдущем примере):

```
with TEXT_IO; use TEXT_IO;
-- Эта строка необходима, так как мы хотим при-
-- своить начальное значение переменной
-- TODAYS_DATE с помощью процедуры GET.
package CHECK_DATES_ALT is
package DAYS_MODULE is
-- Здесь размещаются об'явления из пакета
-- DAYS_MODULE, приведенные ранее.
end DAYS_MODULE;
use DAYS_MODULE;
-- Эта фраза use нужна, так как мы не хотим
```

```

-- пользоваться составными именами.
function IS_VALID_DATE ( FORM_DATE : DATE )
    return BOOLEAN;
-- Это об'явление функции является базисным
-- об'явлением.
procedure FILL_IN_DATE
    (PROC_F_DATE : in out DATE;
     GOOD_DATE   : out BOOLEAN ) ;
    TODAYS_DATE : DATE ;
end CHECK_DATES_ALT
-- Символ ";" превратит эту спецификацию в
-- об'явление пакета.

```

### 7.1.3. Приватные типы

*Приватные типы* Ады предназначены для того, чтобы скрыть от пользователя фактические детали реализации типов. Скрытие этой информации достигается следующим способом: операции, допустимые для объектов приватного типа, ограничены присваиванием и проверкой на равенство и/или неравенство. Любые другие желаемые операции над объектами приватных типов должны быть определены явно в видимой части пакета.

Объявление приватного типа размещается в видимой части пакета, а подробности, касающиеся его реализации, приводятся в приватной части пакета. Объявление приватного типа имеет форму

```
type имя_приватного_типа is private
```

Существует даже еще более жестко ограниченный вид приватного типа, называемый *ограниченным приватным типом* (limited private type). Объявление такого типа имеет форму

```
type имя_ограниченного_приватного_типа is limited private;
```

Для ограниченных приватных типов не разрешены никакие предопределенные операции. Таким образом, присваивание и проверка на равенство/неравенство для них запрещены. Единственным возможным способом применения ограниченных приватных типов может служить использование тех функций и процедур, которые представлены в видимой части пакета и имеют параметры ограниченного приватного типа.

**Пример.** Ниже следует спецификация пакета с объявлениями приватных и ограниченных приватных типов.

```

package ROSTER is
    type STUD_NAME is private ;
    -- Это об'явление приватного типа размещается
    -- в видимой части спецификации пакета.
    -- Пользователи пакета не должны обладать
    -- возможностью изменять фамилии студентов.
    -- Однако, им предоставлено право пользоваться
    -- операциями проверки фамилий студентов на
    -- равенство/неравенство с тем, чтобы они
    -- смогли, например, выяснить, какие курсы
    -- студент изучает.
    type STUD_GRADE is limited private;
    -- Никто (кроме лиц, обладающих особыми полномо-
    -- чиями, как вы вскоре это увидите), не
    -- должен иметь доступ к оценкам студентов.

```

```

-- Операции присваивания и сравнения находят-
-- ся под строгим контролем. Именно по этой
-- причине мы сделали данный тип ограничен-
-- ным приватным, а не просто приватным.
type TEACHER_PASS is private;
-- Для вычисления оценок или выдачи ведомостей
-- требуется пред'явить пароль преподавателя.
procedure GET_PASS
  ( FORM_PASS : out TEACHER_PASS ;
    TEACH_ID   : STRING (1 .. 9) );
type COURSE_INFO is
-- Этот комбинированный тип взят из программы
-- GR_POINT_AVE из разд.4.3.
record
  CR_ID   : STRING (1 .. 4);
  CR_NO   : STRING (1 .. 4);
  CR_CRDT : NATURAL;
end record;
type ROSTER_REC is
record
  ST_NAME : STUD_NAME;
  ST_ID   : STRING (1 .. 9);
  ST_CRS  : COURSE_INFO;
  CRS_GRADE : STUD_GRADE;
end record;
-- Не все сведения о студентах должны быть до-
-- ступны пользователям. Например, при регистрации
-- оценок печатаются не фамилии, а личные номера
-- студентов. Если нужно напечатать оценки студен-
-- тов, то необходимо ввести пароль преподава-
-- теля (см. ниже об'явление процедуры).
function IS_A_VALID_STUDENT
  ( FORM_REC : ROSTER_REC ) return BOOLEAN;
-- Проверка наличия студента в списках. Только
-- студент, фигурирующий в списках, может
-- получать оценку.
procedure COMPUTE_GRADE_N_FILL_NAME
  ( FORM_REC : in out ROSTER_REC ;
    FORM_PASS : in TEACHER_PASS );
-- Эта процедура вносит в списки имя студента
-- (величину приватного типа), вычисляет ито-
-- говую оценку студента и помещает ее в
-- компонент CRS_GRADE.
procedure PRINT_GRADE
  ( FORM_REC : in ROSTER_REC ;
    FORM_PASS : in TEACHER_PASS );
-- Эта процедура выводит оценки, если задан
-- правильный пароль учителя.
private
type STUD_NAME is new STRING (1 .. 20);
type STUD_GRADE is range 0 .. 100 ;
-- STUD_GRADE - это целое число в диапазоне от
-- 0 до 100, но внешний пользователь не имеет
-- доступа к его реализации.
type TEACHER_PASS is range 0 .. 99999 ;
end ROSTER
-- Символ ";" превратит эту спецификацию в
-- об'явление пакета.

```

**Пример.** Ниже показано использование пакета ROSTER (Список) в процедуре SAMPLE\_ROSTER\_USE.

```

with ROSTER; use ROSTER;
with TEXT_IO; use TEXT_IO;
procedure SAMPLE_ROSTER_USE is
  CURR_ROSTER : ROSTER_REC;
  ACT_TEACH_ID : STRING( 1 .. 9 );
  ACT_PASS     : TEACHER_PASS;
  OTHER_ROSTER : ROSTER_REC;
begin
  GET ( CURR_ROSTER.ST_ID );
  GET ( CURR_ROSTER.ST_CRS );
  GET ( ACT_TEACHER_ID );
  GET ( ACT_PASS, ACT_TEACHER_ID );
  -- Вводится личный номер преподавателя, и про-
  -- цедура должна сгенерировать, по всей вероят-
  -- ности, правильный пароль. Значение этого па-
  -- роля передается переменной ACT_PASS ограни-
  -- ченного приватного типа. Пользователю неиз-
  -- вестно действительное значение переменной
  -- ACT_PASS, и он не может изменить его.
  if IS_A_VALID_STUDENT ( CURR_ROSTER )
  then
    COMPUTE_GRADE_N_FILL_NAME ( CURR_ROSTER,
                                ACT_PASS );
    PRINT_GRADE ( CURR_ROSTER, ACT_PASS );
    -- Если пароль, содержащийся в ACT_PASS, пра-
    -- вильный, то вычисляется оценка. Она присваи-
    -- вается CURR_ROSTER.CRS_GRADE, принадлежа-
    -- щей к ограниченному приватному типу.
  end if;
  GET ( OTHER_ROSTER.ST_ID );
  GET ( OTHER_ROSTER.ST_CRS );
  if IS_A_VALID_STUDENT ( OTHER_ROSTER )
  then
    -- Будем считать, что эта функция сверяет лич-
    -- ный номер студента со списком правильных
    -- фамилий студентов и их личных номеров.
    -- Этот список известен только внутри функции.
    then
      COMPUTE_GRADE_N_FILL_NAME ( OTHER_ROSTER,
                                  ACT_PASS );
      if OTHER_ROSTER.ST_NAME /= CURR_ROSTER.ST_NAME
      then
        -- Проверка на равенство/неравенство разра-
        -- шена для приватных типов. Но было бы оши-
        -- бкой записать отношение
        -- OTHER_ROSTER.CRS_GRADE /=
        -- CURR_ROSTER.CRS_GRADE,
        -- так как в нам используются ограниченные
        -- приватные типы. Нельзя написать оператор
        -- OTHER_ROSTER.CRS_GRADE :=
        -- CURR_ROSTER.CRS_GRADE ;
        -- так как присваивание значений ограничен-
        -- ного приватного типа запрещается. Однако
        -- можно написать:
        -- OTHER_ROSTER.ST_NAME :=
        -- CURR_ROSTER.ST_NAME;
      end if;
    end if;
  end if;
end;

```

```

    then
      PUT ( " Different names " );
    end if;
  end if;
end SAMPLE_ROSTER_USE;

```

Внешний пользователь частного типа не имеет сведений о реализации этого типа. Даже если пользователь догадается, что фамилия студента представлена в виде строки символов, то он все равно не сможет записать оператор присваивания

```
OTHER_ROSTER.ST_NAME := "12345678901234567890";
```

поскольку два приведенных здесь объекта принадлежат к разным типам. Преобразование же типа в данном случае запрещено.

Скрытие информации достигается здесь при помощи введения частных типов. Оно вполне оправдано в пакете ROSTER с точки зрения обеспечения сохранности сведений и позволяет защитить ведомости успеваемости студентов от подделки. Кроме того, сокрытие информации дает возможность писать прикладные программы, не зависящие от конкретного способа реализации частного типа. Пусть, например, частный тип STUD\_GRADE (Студ. оценка) реализован как целый тип с диапазоном значений от 0 до 100. Можно изменить эту реализацию и сделать тип этой переменной; скажем, перечисляемым или строковым. При этом не потребуются делать никаких изменений в прикладных программах, использующих пакет, в котором определен этот тип. Подпрограммы, в которых использовались формальные параметры частных типов, придется все же переписать заново.

Скрытие информации помогает при создании и локализации новых типов данных. Это иллюстрирует следующий пример.

**Пример.** Объявление пакета GIANT\_INT\_ARITHMETIC (Для цел арифм) определяет арифметические операции для целых чисел с большим количеством цифр, входящих в их состав.

```

package GIANT_INT_ARITHMETIC is
  type GIANT_INTEGER is private;
  function INT_TO_GIANT ( INT_FORM : INTEGER )
    return GIANT_INTEGER;
  -- Эта функция вырабатывает значение типа
  -- GIANT_INTEGER по значению переменной
  -- INT_FORM, выровненному справа по
  -- восьмой компоненте.
  function GIANT_TO_INT ( GIANT_FORM :
    GIANT_INTEGER ) return INTEGER;
  -- Эта функция будет пытаться преобразовать
  -- длинное целое число в обычное, если
  -- только оно будет не слишком велико.
  function GIANT_ADD ( LEFT_FORM, RIGHT_FORM :
    GIANT_INTEGER ) return GIANT_INTEGER;
  -- Эта функция складывает два длинных целых
  -- числа.
  function GIANT_SUB ( LEFT_FORM, RIGHT_FORM :
    GIANT_INTEGER ) return GIANT_INTEGER;
  -- Эта функция выполняет вычитание длинных
  -- целых чисел.
  function GIANT_LT ( LEFT_FORM, RIGHT_FORM :
    GIANT_INTEGER ) return BOOLEAN;
  -- Эта функция вырабатывает значения TRUE,
  -- если LEFT_FORM меньше, чем RIGHT_FORM.

```



```

function GIANT_GT ( LEFT_FORM, RIGHT_FORM :
    GIANT_INTEGER ) return BOOLEAN;
    -- Эта функция вырабатывает значение TRUE,
    -- если LEFT_FORM больше, чем RIGHT_FORM.
function GIANT_TO_STRING ( GIANT_FORM :
    GIANT_INTEGER ) return STRING;
    -- Эта функция преобразует длинное целое
    -- число в строку.
function STRING_TO_GIANT ( STRING_FORM :
    STRING ) return GIANT_INTEGER;
    -- Эта функция преобразует строку в длинное
    -- целое число.
private
type GIANT_INTEGER is array ( 1 .. 8 )
    of INTEGER;
    -- Это объявление можно заменить на другое,
    -- например, так:
    -- type GIANT_INTEGER is
    --     record
    --         FIRST_HALF : INTEGER;
    --         SECOND_HALF : INTEGER;
    --     end record;
end GIANT_INT_ARITHMETIC;

```

**Пример.** Теперь проиллюстрируем употребление пакета GIANT\_INT\_ARITHMETIC для работы с длинными целыми числами на примере подпрограммы GIANT\_INT\_USE.

```

with GIANT_INT_ARITHMETIC;
use GIANT_INT_ARITHMETIC;
with TEXT_IO; use TEXT_IO;
procedure GIANT_INT_USE is
    I, J : INTEGER := 55;
    I_BIG, J_BIG : GIANT_INTEGER;
    I_STRING, J_STRING : STRING(1 .. 13) :=
        "1234567890123";
begin
    I_BIG := INT_TO_GIANT(I);
    J_BIG := INT_TO_GIANT(J);
    -- Здесь выполняется преобразование обычных
    -- целых чисел в длинные целые числа.
    I_BIG := GIANT_ADD(I_BIG, J_BIG);
    -- Здесь два длинных целых числа складываются.
    I := GIANT_TO_INT(I_BIG);
    -- Здесь длинное целое число преобразуется
    -- обратно в обычное целое.
    I := I ** 2;
    I_BIG := INT_TO_GIANT(I);
    J_BIG := STRING_TO_GIANT(J_STRING);
    -- Здесь строка преобразуется в длинное
    -- число.
    J_BIG := GIANT_SUB(J_BIG, I_BIG);
    -- Это пример вычитания одного длинного
    -- целого числа из другого.
    if GIANT_LT(I_BIG, J_BIG)
    -- В этой строке выполняется проверка
    -- условия "меньше или равно" для двух
    -- длинных целых чисел.

```

```

then
  I_BIG := STRING_TO_GIANT(I_STRING);
else
  I_BIG := GIANT_ADD(J_BIG, J_BIG);
end if;
if GIANT_GT(I_BIG, J_BIG)
  -- В этой строке выполняется проверка
  -- условия "больше" для двух длинных
  -- целых чисел.
  then
    I_STRING := GIANT_TO_STRING(I_BIG);
    -- Здесь длинное целое число преобразует-
    -- в строку.
  else
    I_STRING := GIANT_TO_STRING(J_BIG);
  end if;
  PUT(I_STRING);
end GIANT_INT_USE;
```

Спецификация пакета `GIANT_INT_ARITHMETIC` описывает операции для так называемого *абстрактного типа данных*. Абстрактный тип данных локализован внутри пакета, т. е. пользователю ничего не известно о том, как этот абстрактный тип данных реализован.

#### 7.1.4. Использование пакетов

Приведенные до сих пор спецификации демонстрируют некоторые типичные области применения пакетов. Вот список этих областей в порядке возрастания их сложности:

- использование пакетов как совокупностей взаимосвязанных типов данных и объявлений объектов (пример – пакет `DAYS_MODULE`). Такие пакеты экспортируют объекты и типы;
- использование пакетов как совокупностей взаимосвязанных подпрограмм (пример – пакет `CHECK_DATES`) и иных программных сегментов, т. е. пакетов и задач. Эти пакеты экспортируют программные сегменты;
- использование пакетов как средств, реализующих абстрактные типы и множество операций, разрешенных для значений, принадлежащих к этим типам (пример – пакет `GIANT_INT_ARITHMETIC`). Такие пакеты экспортируют как типы, так и программные сегменты;
- последней (четвертой) сферой применения может служить использование пакетов, которые экспортируют объекты, типы, программные сегменты и, вдобавок, выполняют внутри самого пакета регистрацию его внутренних состояний. Следующий пример иллюстрирует это.

**Пример.** Спецификация пакета, предназначенная для работы с индексами, отражающими деловую активность на бирже:

```

package MARKET is
  type DIRECTION is
    (DOWN, UNCHANGED, UP);
  type AVERAGE_INDICATORS is private;
  -- Эти типы можно реализовать как
  -- массивы скользящих средних значений,
  -- вычисленных, например, за последние
  -- 5, 10 или 20 дней.
  type OSCILLATORS is private;
```

```

-- Эти типы могут быть представлены
-- как массивы индикаторов текущего
-- состояния деловой активности.
-- Об'екты, относящиеся к этим типам,
-- будут иметь значения, отражающие
-- скорость изменения уровня деловой
-- активности.
-- *** MARK ***
type LEVEL is digits 8 range 0.00 ..
                                10000.00;
function CURR_LEVEL return LEVEL;
-- Текущее состояние рынка отслежива-
-- ется внутренними средствами пакета,
-- такими, например, как текущее зна-
-- чение переменной DOW_JONES (Индекс
-- Доу-Джонса.)
function NEXT_GUESS(FORM_LEVEL : LEVEL;
                    PREV_FORM_MOVE : DIRECTION;
                    FORM_OSCILL : OSCILLATOR;
                    FORM_AVER : AVERAGE_INDICATORS);
-- По значениям формальных параметров эта
-- функция вырабатывает определенный
-- прогноз состояния рынка.
return LEVEL;
procedure UPDATE_LEVEL(FORM_LEVEL : in out
                      LEVEL;
                      NEW_MOVE : DIRECTION;
                      FORM_MAGN : LEVEL);
-- Новый уровень рынка получается с учетом
-- старого его состояния и новых изменений.
private
type AVERAGE_INDICATORS is array (1 .. 10)
                                of FLOAT;
type OSCILLATORS is array (1 .. 3)
                          of FLOAT;
-- ??? MARK ???
end MARKET -- Если добавить символ ";", то
-- получится законченное об'явление
-- пакета.

```

Спецификация пакета MARKET (Рынок) экспортирует объявления (например, объявление типа AVERAGE\_INDICATORS) и подпрограммы (например, процедуру UPDATE\_LEVEL). Пакет также будет отслеживать уровень деловой активности с помощью внутренних средств.

Пакеты Ады, встречающиеся на практике, несут в себе сочетания особенностей всех описанных выше четырех видов пакетов.

### 7.1.5. Отложенные константы

В заключение отметим, что константы приватного типа можно объявлять в видимой части пакета как так называемые *отложенные константы*. Значения отложенных констант указываются в приватной части. Пользователь пакета не знает действительного значения отложенной константы, но он может присваивать ее значение другим приватным объектам или сравнивать ее с ними.

Объявление отложенной константы имеет форму

имя\_константы: constant тип\_или\_подтип;

Например, вместо строки с пометкой **\*\*\* MARK \*\*\*** в спецификации пакета **MARKET** можно поместить следующее объявление константы:

```
STANDARD_OSCILLATOR: constant OSCILLATORS;
```

Фактическое значение константы можно задать с помощью объявления

```
STANDARD_OSCILLATOR : constant := (0.05, 0.55, 0.95);
```

которое следует поместить вместо строки с пометкой **??? MARK ???**

## 7.2. ТЕЛА ПАКЕТОВ

Реализация пакета осуществляется при помощи *тела пакета*. Введем два вида тел пакетов. Первый вид содержит только объявления:

```
package body имя_пакета is
-- Возможные объявления.
end имя_пакета;
```

Второй вид содержит член «последовательность\_операторов»:

```
package body имя_пакета is
-- Возможные объявления.
begin
-- Последовательность_операторов.
end имя_пакета;
```

Еще один вид тел пакетов используется в связи с исключительными ситуациями. Он будет рассмотрен в гл. 11.

Имя\_пакета здесь должно быть идентично имени, указанному в спецификации пакета. Указывать имя\_пакета после зарезервированного слова **end** необязательно. Каждое объявление известно в соответствующей спецификации пакета и может быть использовано в теле этого пакета. Обратное утверждение будет неверным, т. е. объявления, сделанные в теле пакета, неизвестны и не могут использоваться в его спецификации, так как они невидимы вне тела пакета.

Спецификации и тела пакетов можно транслировать отдельно. В этом случае трансляция спецификации пакета должна предшествовать трансляции его тела.

Вот простой пример тела пакета:

```
package body DAYS_MODULE is
-- Здесь нет ни объявлений, ни последовательности_операторов.
end DAYS_MODULE;
```

Это тело пакета соответствует спецификации пакета **DAYS\_MODULE**, представленной в начале главы. Ввиду того что пакет **DAYS\_MODULE** содержит лишь объявления типов и объектов, наличие тела у этого пакета не требуется. В тех случаях, когда пакет представляет собой только набор типов, констант и переменных, тело для пакета не обязательно.

Однако наличие тела пакета обязательно требуется в том случае, если его спецификация содержит объявления подпрограмм или пакетов. Тела этих пакетов и подпрограмм должны образовывать часть тела данного пакета.

**Пример.** Тело пакета, соответствующее спецификации пакета **CHECK\_DATES\_ALT**, таково:

```
with TEXT_IO; use TEXT_IO;
package body CHECK_DATES_ALT is
  LEAP_YEAR : BOOLEAN;
  INPUT_LEAP : INTEGER;
```

```

package body DAYS_MODULE is
  -- Здесь должно располагаться тело
  -- пакета DAYS_MODULE.
end DAYS_MODULE;
function IS_VALID_DATE (FORM_DATE:DATE);
  return BOOLEAN is
  -- Здесь должно размещаться тело
  -- функции. Оно идентично телу
  -- одноименной функции из программы
  -- ACCR_INTEREST.
end IS_VALID_DATE;
procedure FILL_IN_DATE
  (PROC_F_DATE : in out DATE;
   GOOD_DATE   : out BOOLEAN) is
  -- Сюда нужно вставить тело процедуры.
  -- Оно идентично телу одноименной
  -- процедуры из программы ACCR_INTEREST.
end FILL_IN_DATE;
end CHECK_DATES_ALT;

```

Тело пакета CHECK\_DATES\_ALT из предыдущего примера имеет декларативную часть, но не содержит последовательности операторов. Если же последовательность операторов присутствует, то она может выполнять, например, инициализацию некоторых переменных пакета. Следующий пример иллюстрирует этот процесс.

**Пример.** Вот другая версия тела пакета CHECK\_DATES\_ALT:

```

package body CHECK_DATES_ALT is
  -- Сюда следует поместить все об'явления
  -- из тела пакета предыдущего примера.
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
begin
  PUT(" Enter today's date as YYMMDD ");
  NEW_LINE;
  GET(TODAYS_DATE.YEAR_NO, 2);
  GET(TODAYS_DATE.MONTH_NO, 2);
  GET(TODAYS_DATE.DAY_NO, 2);
end CHECK_DATES_ALT;

```

Структура этой версии пакета CHECK\_DATES\_ALT показана на рис. 7.1.

Приведем еще примеры спецификаций и тел пакетов для ряда программ из предыдущих глав.

**Пример.** Некоторые типы из программы YIELD\_COMPUTATION (см. гл. 6) и функцию «-» можно свести в один пакет, предназначенный для вычисления дохода. Пакет имеет следующий вид:

```

with CHECK_DATES_ALT;
package SECURITIES is
  use CHECK_DATES_ALT;
  type INTEREST is digits 13;
  type DISCOUNT_INTEREST is new INTEREST;
  type AT_MAT_INTEREST is new INTEREST;
  type INTEREST_KIND is (COUPON, AT_MATURITY,
                        DISCOUNT);
  type DAY_COUNT_BASIS is (ACT_ACT, ACT_360,
                          M_30_Y_360 );

```

## PACKAGE CHECK\_DATES\_ALT

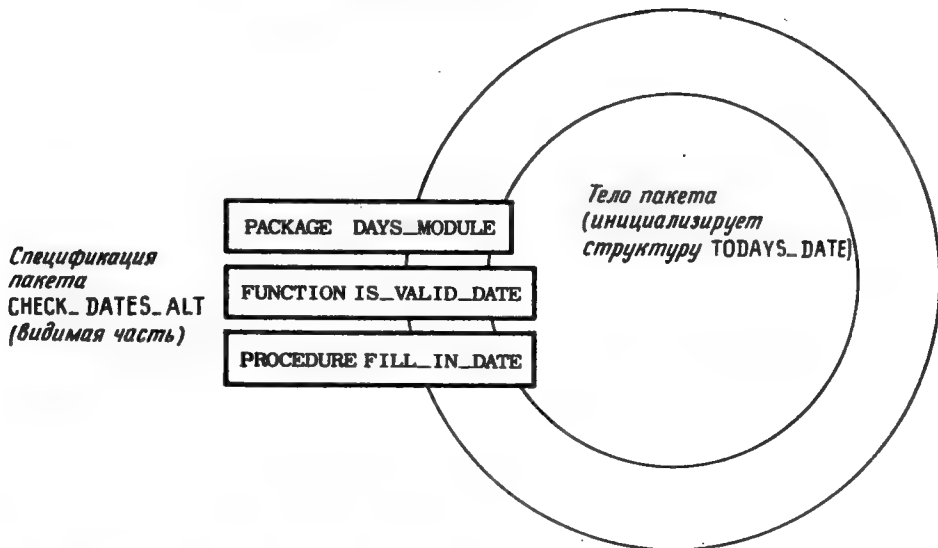


Рис. 7.1. Элементы пакета CHECK\_DATES\_ALT.

```

type PRICES is digits 13;
type DISCOUNT_PRICES is new PRICES;
type SECURITY ( INT_PAYM , INTEREST_KIND ) is
  record
    SEC_NAME : STRING ( 1 .. 10 );
    SETL_DATE : DATE;
    MAT_DATE : DATE;
    DAY_KIND : DAY_COUNT_BASIS;
    YLD : INTEREST;
    case INT_PAYM is
      when AT_MATURITY =>
        ISSUE_DATE : DATE;
        S_RATE : AT_MAT_INTEREST;
        QT_PRICE : PRICES;
      when DISCOUNT =>
        DISC_PRICE : DISCOUNT_PRICES;
        DISC_RATE : DISCOUNT_INTEREST;
      when others => null;
    end case;
  end record;
function '-' (X, Y : DATE ) return NATURAL;
and SECURITIES;

```

Соответствующее тело пакета таково:

```

package body SECURITIES is
function "-" (X, Y : DATE )
-- Эта процедура подсчитывает количество прошед-
-- ших дней в предположении, что в месяце 30
-- дней, а в году - 360 (M_30_Y_360).
return NATURAL is

```

```

begin
-- Сюда следует поместить тело функции, см. гл.6.
and "-";
end SECURITIES;

```

Другой полезный пакет, отслеживающий неприсутственные дни, приведен в следующем примере. Объявления и тела взяты в основном из программы ACCR\_INTEREST (см. гл. 5).

**Пример.** В состав этого пакета входит последовательность операторов, выполняющая инициализацию списка праздников. Если потребуется изменить этот список, то можно соответствующим образом модифицировать текст тела пакета, а затем перекомпилировать его.

```

with CHECK_DATES_ALT;
package LEGAL_HOLIDAYS is
  use CHECK_DATES_ALT;
  function IS_LEGAL_HOLIDAY(FORM_DATE : DATE)
    return BOOLEAN;
end LEGAL_HOLIDAYS;
-- Здесь заканчивается спецификация.
package body LEGAL_HOLIDAYS is
  NO_OF_HOLIDAYS : NATURAL := 2;
  NO_GOOD_HOLIDAYS : NATURAL;
  GOOD_HOLIDAY : BOOLEAN;
  type HOLIDAYS is array (1 .. NO_OF_HOLIDAYS)
    of DATE;
  ACT_HOLIDAYS : HOLIDAYS :=
    -- Проинициализируем эту переменную
    -- предполагаемыми значениями дат
    -- праздников. Здесь принимаются во
    -- внимание только компоненты даты
    -- MONTH_NO, DAY_NO и YEAR_NO.
    ((MONDAY, JANUARY, 7, 4, 1, 1, 1985),
     (MONDAY, JANUARY, 1, 1, 1, 1, 1986));
  function IS_LEGAL_HOLIDAY(FORM_DATE:DATE)
    return BOOLEAN is
  begin
    for I in 1 .. NO_GOOD_HOLIDAYS
      loop
        if FORM_DATE.TOTAL_DAYS =
          ACT_HOLIDAYS(I).TOTAL_DAYS
        then
          return TRUE;
        end if;
      end loop;
    return FALSE;
  end IS_LEGAL_HOLIDAY;
begin
-- Далее располагается последовательность
-- операторов данного пакета. Эти операторы
-- опять взяты из программы ACCR_INTEREST
-- из гл.5.
  NO_GOOD_HOLIDAYS := 0;
  for I in 1 .. NO_OF_HOLIDAYS
    loop
      NO_GOOD_HOLIDAYS := NO_GOOD_HOLIDAYS + 1;
      FILL_IN_DATE(ACT_HOLIDAYS(NO_GOOD_HOLIDAYS),

```

```

GOOD_HOLIDAY);
if not GOOD_HOLIDAY
then
  NO_GOOD_HOLIDAYS := NO_GOOD_HOLIDAYS - 1;
end if;
end loop;
end LEGAL_HOLIDAYS;

```

Вспомните, что ресурсы, объявленные в теле пакета (например, переменные NO\_GOOD\_HOLIDAYS и NO\_OF\_HOLIDAYS), неизвестны и недоступны за пределами этого тела.

Последним пакетом данного раздела будет пример пакета для программы NAME\_PHONE из гл. 4. Он позволяет выполнять некоторые полезные операции с текстами.

**Пример.** Этот пакет для программы NAME\_PHONE назван LINE\_HANDLING (обработка строк).

```

package LINE_HANDLING is
  -- Член COMMA (запятая) отброшен, и пере-
  -- менная WRK_LINE стала компонентом струк-
  -- туры T_REC.
  type LINE_LENGTH is INTEGER range 0 .. 255;
  type T_REC (LINE_LN : LINE_LENGTH) is
    record
      WRK_LINE : STRING(1 .. T_LENGTH);
    end record;
  procedure LOW_TO_UPPER(F_REC : in out T_REC);
  procedure IGNORE_LEADING_SPACES(
    STRT_POS : out NATURAL;
    F_REC : in out T_REC);
  procedure FIND_NEXT_SP_OR_COMMA(
    STRT_POS : in out NATURAL;
    END_POS : in out NATURAL;
    F_REC : in out T_REC);
  procedure PLACE_SPACES(
    STRT_POS : in out NATURAL;
    END_POS : in out NATURAL;
    F_REC : in out T_REC);
end LINE_HANDLING;

```

Соответствующее тело пакета, во многом совпадающее с текстом из программы NAME\_PHONE, таково:

```

package body LINE_HANDLING is
  procedure LOW_TO_UPPER(F_REC : in out T_REC)
  is
  begin
    for I in 1 .. F_REC.LINE_LN
    loop
      case F_REC.WRK_LINE(I) is
        when 'a' .. 'z' =>
          F_REC.WRK_LINE(I) := CHARACTER'VAL(
            CHARACTER'POS('A') -
            CHARACTER'POS('a') +
            CHARACTER'POS(WRK_LINE(I)) );
        when others => NULL;
      end case;
    end loop;
  end LOW_TO_UPPER;

```



```

    end loop;
  and LOW_TO_UPPER;
  procedure IGNORE_LEADING_SPACES(
    STRT_POS : out NATURAL;
    F_REC    : in out T_REC) is
  begin
    for I in 1 .. F_REC.LINE_LN
      loop
        STRT_POS := I;
        exit when F_REC.WRK_LINE(I) /= ' ';
      end loop;
  and IGNORE_LEADING_SPACES;
  procedure FIND_NEXT_SP_OR_COMMA(
    STRT_POS : in out NATURAL;
    END_POS  : in out NATURAL;
    F_REC    : in out T_REC) is
    -- Эта процедура присваивает переменной
    -- END_POS значение номера позиции послед-
    -- него символа, отличающегося от пробела
    -- или запятой и располагающегося после
    -- позиции строки STRT_POS.
  begin
    END_POS := STRT_POS;
    for I in STRT_POS .. F_REC.LINE_LN
      loop
        exit when
          F_REC.WRK_LINE(I) = ' ' or
          F_REC.WRK_LINE(I) = ',';
        END_POS := I;
      end loop;
  and FIND_NEXT_SP_OR_COMMA;
  procedure PLACE_SPACES(
    STRT_POS : in out NATURAL;
    END_POS  : in out NATURAL;
    F_REC    : in out T_REC) is
    -- Эта процедура заменяет на пробелы символы,
    -- расположенные между позициями строки
    -- STRT_POS и END_POS, и заменяет на пробел
    -- первую обнаруженную запятую.
  begin
    for I in STRT_POS .. END_POS
      loop
        F_REC.WRK_LINE(I) := ' ';
      end loop;
    for I in END_POS + 1 .. LINE_LN
      loop
        IF F_REC.WRK_LINE(I) = ','
        then
          F_REC.WRK_LINE(I) := ' ';
          exit;
        end if;
        exit when F_REC.WRK_LINE(I) /= ' ';
      end loop;
    end PLACE_SPACES;
  and LINE_HANDLING;

```

Применение пакета LINE\_HANDLING демонстрируется в следующем примере, который иллюстрирует задачу проверки входных данных.

**Пример.** Предположим, что на вход программы поступают целые числа, возможно, по несколько штук в строке. Следующая программа будет печатать текст Valid integer (Корректное целое число) или Invalid integer (Ошибочное целое число) для каждой группы символов, не включающих пробел, по бокам которой располагаются по крайней мере по одному пробелу или запятой. (Повторите определение целых литералов, оно дано в гл. 1.) Проверяются только десятичные целые числа. Признак конца данных — XX.

```
with LINE_HANDLING;
use LINE_HANDLING;
with TEXT_IO; use TEXT_IO;
procedure CHECK_FOR_INTEGERS is
  CURR_REC : T_REC;
  -- Обеспечивается доступность и непосред-
  -- ственная видимость типа T_REC из пак-
  -- та LINE_HANDLING.
  INPUT_LN : NATURAL;
  -- Эта переменная задает длину строки.
  INP_LINE : STRING(1 .. INPUT_LN);
  CURR_ST : NATURAL;
  CURR_END : NATURAL;
  END_A_NO : NATURAL;
  A_NUMBER : STRING(1 .. END_A_NO);
  function CHECK_NUMBER(F_STRING : STRING;
                        F_END : NATURAL)
    return BOOLEAN is
  begin
    for I in 1 .. F_END
      loop
        case F_STRING(I) is
          when '0' .. '9' => NULL;
          when '-' | '+' =>
            -- Только первый символ может
            -- быть знаком.
            if I /= 1
              then
                return FALSE;
            end if;
          when '_' =>
            -- Наличие символа подчеркивания в числе
            -- разрешено, но этот символ должен быть
            -- окружен цифрами или другими символами
            -- подчеркивания.
            if I = 1 or I = F_END or else
              not(F_STRING(I-1) in '0' .. '9' or
                 F_STRING(I-1) = '_') or else
              not(F_STRING(I+1) in '0' .. '9' or
                 F_STRING(I+1) = '_')
            then
              return FALSE;
            end if;
          when 'E' =>
            -- Если присутствует показатель степени,
            -- то он должен быть окружен цифрами.
```

```

        if I = 1 or I=F_END or else
        F_STRING(I-1) not in '0' .. '9' or else
        F_STRING(I+1) not in '0' .. '9'
        then
            return FALSE;
        end if;
        when others => return FALSE;
    end case;
end loop;
return TRUE;
end CHECK_NUMBER;
begin
    GET_LINE(INP_LINE, INPUT_LN);
    while INP_LINE /= "XX"
    loop
        CURR_REC := (INPUT_LN, INP_LINE);
        -- Переменной CURR_REC присваивается значение аргумента.
        IGNORE_LEADING_SPACES(CURR_ST, CURR_REC);
        LOW_TO_UPPER(CURR_REC);
        while CURR_ST /= CURR_REC.LINE_LN
        loop
            FIND_NEXT_SP_OR_COMMA(CURR_ST, CURR_END,
                                   CURR_REC);
            END_A_NO := CURR_END - CURR_ST;
            A_NUMBER := CURR_REC.WRK_LINE(CURR_ST ..
                                           CURR_END - 1);
            -- Это - присваивание вырезки.
            PUT(A_NUMBER);
            if CHECK_NUMBER(A_NUMBER, END_A_NO) = TRUE
            then
                PUT(" Valid number ");
            else
                PUT(" Invalid number ");
            end if;
            NEW_LINE;
            PLACE_SPACES(CURR_ST, CURR_END, CURR_REC);
        end loop;
        GET_LINE(INP_LINE, INPUT_LN);
    end loop;
end CHECK_FOR_INTEGERS;

```

Обработка объявления пакета включает вначале обработку его спецификации, а затем тела. Обработка спецификации пакета состоит из обработки видимой части, за которой следует обработка приватной части. Обработка тела пакета состоит из обработки декларативной части и выполнения последовательности операторов. Может показаться некорректным, но тем не менее это справедливо: обработка тела пакета включает выполнение его последовательности операторов<sup>1)</sup>.

<sup>1)</sup> Термин «обработка» (elaboration) в Аде применяется для обозначения действий, производимых с объявлениями. Термин «выполнение» (execution) обычно относится к исполняемым операторам языка, но не к объявлениям. Поэтому автор в данном случае обращает внимание читателей на некоторую терминологическую несогласованность. — *Прим. перев.*

### 7.3. ПРАВИЛА ВИДИМОСТИ ДЛЯ ПАКЕТОВ

В гл. 6 были даны определения области действия и видимости идентификаторов для подпрограмм и блоков. Теперь к ним будут добавлены правила, регулирующие видимость и область действия для пакетов.

В разд. 7.1 упоминалось, что видимая часть спецификации пакета экспортируется в другие программные сегменты. После того как ресурсы из видимой части будут экспортированы, например, при помощи объявлений пакетов или благодаря употреблению фразы подключения контекста *with*, эти ресурсы получают область действия, простирающуюся до конца пакета, подпрограммы или блока, которые их импортируют. Видимость экспортированных ресурсов зависит от возможного «заслонения» их объявлений в некоторых участках области действия (см. гл. 6). Если экспортируемые ресурсы невидимы непосредственно, то для видимых величин нужно употреблять составные имена. Непосредственная же видимость, когда можно пользоваться простыми именами, обеспечивается при помощи фразы использования *use*.

Объявления находятся в пределах спецификации и тела пакета. При этом они могут появляться в видимой части спецификации пакета, в приватной части спецификации пакета и в теле пакета. Действие видимых объявлений пакета в его пределах распространяется на приватную часть пакета и на его тело. Действие объявлений, расположенных в приватной части, распространяется и на тело пакета, а объявления, расположенные в теле пакета, действуют только в пределах этого тела. Ясно, что нельзя ссылаться на ресурс за пределами его области действия.

**Пример.** Здесь показано применение правил, касающихся области действия и видимости.

```

procedure SCOPE_N_VISIBILITY
-- Далее располагается спецификация пакета.
LL : INTEGER := 0;
package AA is
  II : INTEGER;
  -- Область действия идентификаторов II
  -- и JJ охватывает спецификацию и тело
  -- пакета AA. Она также охватывает
  -- процедуру SCOPE_N_VISIBILITY, начи-
  -- ная с этой точки и до конца процедуры.
  type JJ is private;
  function CC(K_F : INTEGER) return JJ;
private
  type JJ is new INTEGER;
  -- Область действия и видимость иденти-
  -- фикатора JJ начинаются здесь и закан-
  -- чиваются в конце тела пакета AA.
end AA;
-- Далее располагается тело пакета.
package body AA is
  KK := INTEGER := 5;
  -- Область действия и видимость иденти-
  -- фикатора KK начинаются здесь и заканчи-
  -- ваются в конце тела пакета AA.
  -- Идентификатор KK не видим за пределами
  -- данного тела пакета.
  function CC (K_F : INTEGER) return JJ is
  begin
    if K_F > KK
    then

```

```

        return JJ(2*K_F);
        -- Здесь используется преобразование
        -- величины к производному типу.
    else
        return JJ(K_F + KK);
    end if;
end CC;
end AA;
use AA;
MM, NN : JJ;
-- Переменные MM и NN относятся к приватному
-- типу JJ.
begin
    MM := CC(LL);
    II := LL;
    -- Переменная II видима непосредственно.
    NN := MM;
    -- Присваивание для об'ектов приватных типов
    -- разрешается.
    PUT(" now NN and MM are equal ");
    -- Далее располагается составной оператор
    -- блока.
    declare
        II : INTEGER := 13;
        -- Область действия этого идентификатора
        -- II и его видимость начинаются в этой
        -- точке и заканчиваются в конце блока.
        -- Идентификатор II, об'явленный в пакете
        -- AA, теперь заслонен. Но его можно сде-
        -- лать видимым, если использовать обо-
        -- значение AA.II.
    begin
        NN := CC(II);
        if MM = NN
            then PUT(" equal ");
            else PUT(" unequal ");
        end if;
    end;
    -- Конец блока.
    -- Те же самые операторы, расположенные
    -- вне блока, дали бы иной результат.
    NN := CC(II);
    if MM = NN
        then PUT(" second equal ");
        else PUT(" second unequal ");
    end if;
end SCOPE_N_VISIBILITY;

```

## 7.4. ОБЪЯВЛЕНИЯ ПЕРЕИМЕНОВАНИЯ

К этому моменту мы уже встретились с достаточно большим количеством программ, в которых использовались составные имена. Если отсутствует фраза использования `use`, то все видимые идентификаторы можно использовать только с помощью составных имен. Если же будет несколько уровней вложенности пакетов, то запись составных идентификаторов станет весьма громоздкой и неудобной. Более того,

иногда не рекомендуется употреблять фразу *use* вообще, чтобы избежать возможной путаницы и конфликтов между именами. В таких ситуациях объявление переименования дает возможность ввести сокращенную форму обозначения для многоуровневых составных имен.

Здесь будут введены три вида *объявлений переименования* (*renaming declarations*). Первый из них переименовывает объекты, второй – спецификации подпрограмм, а третий – пакеты. Прочие виды объявлений переименования, затрагивающие, например, исключительные ситуации, будут рассмотрены в гл. 11.

Первый вид объявлений переименования имеет форму

идентификатор : тип\_или\_подтип renames имя\_объекта;

Например, в последней версии пакета CHECK\_DATES\_ALT (разд. 7.2) можно записать такое объявление переименования:

CURRENT\_MONTH : MONTH\_INT renames TODAYS\_DATE . MONTH\_NO;

Объявления переименования для спецификаций подпрограмм имеют структуру

новое\_имя\_подпрограммы renames старое\_имя\_подпрограммы;

Например, если в подпрограмме объявлен пакет LINE\_HANDLING, то можно воспользоваться таким объявлением:

L\_TO\_U renames LINE\_HANDLING . LOW\_TO\_UPPER;

Третий вид объявлений переименования принимает форму

package новое\_имя\_пакета renames старое\_имя\_пакета;

Пусть, например, имеется такое объявление пакета:

```
package TEXT_MANIP is
package LINE_HANDLING is

and LINE_HANDLING;
-- Здесь даются прочие об'явления.
and TEXT_MANIP;
```

Тогда можно записать следующее объявление, переименовывающее пакет:

package NEW\_LINE\_HAND renames TEXT\_MANIP. LINE\_HANDLING;

Объявление переименования может оказаться неоднозначным, поэтому нужно внимательно следить за употреблением введенных «псевдонимов».

## 7.5. ВВЕДЕНИЕ В РОДОВЫЕ ПАКЕТЫ

Поскольку Ада – язык со строгой типизацией, типы должны быть выражены явно к тому моменту, когда программы, в которых они используются, начинают компилироваться. Кроме того, существуют строгие правила, касающиеся согласования фактических и формальных параметров. Поэтому для написания подпрограммы, которая была бы достаточно универсальна, чтобы работать с множеством типов, уже рассмотренные нами выразительные средства Ады не обеспечивают должной гибкости. В связи с этим строгость правил использования типов в Аде дополняется ее *родовыми средствами* (*generic facilities*), которые позволяют создавать некие трафареты, или модели, пакетов и подпрограмм.

Программы, написанные с применением родовых средств, не годятся для непосредственного выполнения. Поэтому перед выполнением производится конкретизация родовых пакетов или подпрограмм. При *конкретизации* (*instantiation*) создается вполне определенная работоспособная версия пакета или подпрограммы. Эта конкретная версия генерируется с использованием некоторых фактических параметров

(фактических типов). Выполняется согласование фактических параметров с родовыми. Родовые параметры играют роль, в какой-то степени сходную с ролью формальных параметров подпрограммы.

Общая форма *родового объявления* такова:

generic объявление\_родового\_параметра;

Обычный пакет можно превратить в родовой, если поместить перед ним родовое объявление.

В пакете TEXT\_IO (см. приложение В) содержится ряд родовых пакетов. Например, в его состав входит родовой пакет INTEGER\_IO, который имеет следующий вид:

```
generic
  type NUM is range <> ;
  -- Данная строка представляет собой родовое
  -- объявление. Это - объявление родового
  -- параметра. Здесь NUM - родовой параметр
  -- типа, который согласовывается с любым
  -- целым типом. Любой фактический параметр,
  -- представляющий собой имя целого типа,
  -- может быть согласован с NUM.
package INTEGER_IO is
  -- Здесь размещаются объявления об'ектов
  -- и подпрограмм.
end INTEGER_IO;
```

Во многих предыдущих программах были использованы конкретизации родового пакета INTEGER\_IO, при этом применялась такая схема:

```
package имя_пакета is new имя_родового_пакета (один или более родовых факти-
ческих параметров);
```

Например, если AGE - целый тип, то конкретизация может принять вид

```
package INT_IO is new INTEGER_IO (AGE);
```

В этой конкретизации фактический родовой параметр - это целый тип AGE. Предпринимать конкретизацию этого пакета не с целым типом было бы ошибкой.

После конкретизации пакета INTEGER\_IO с использованием фактического параметра AGE реализуется пакет INT\_IO. При этом в каждом объявлении, содержащемся в спецификации пакета INTEGER\_IO, родовой параметр NUM заменяется на AGE. Процесс конкретизации иллюстрирует рис. 7.2.

Перечислим некоторые другие родовые пакеты, входящие в состав пакета TEXT\_IO. Они также использовались в приведенных программах. Ниже следует пакет, предназначенный для ввода-вывода чисел с плавающей точкой:

```
generic
  type NUM is digits <> ;
  -- Данная строка представляет собой родовое
  -- объявление. Это - объявление родового
  -- параметра. Здесь NUM - родовой параметр
  -- типа, который согласовывается с любым
  -- плавающим типом. Любой фактический пара-
  -- метр, представляющий собой имя плавающей-
  -- го типа, может быть согласован с NUM.
package FLOAT_IO is
  -- Содержимое этой части приведено в
  -- приложении В.
end FLOAT_IO;
```

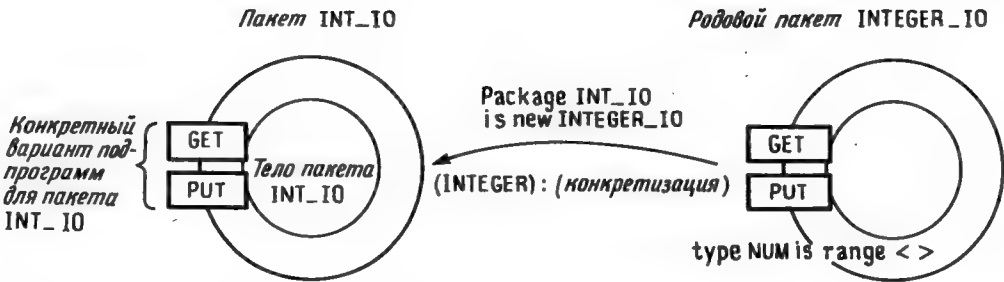


Рис. 7.2. Конкретизация пакета INTEGER\_IO.

Теперь вниманию читателей предлагается описание средств пакета TEXT\_IO, предназначенных для работы с фиксированными типами:

```
generic
  type NUM is delta <> ;
  -- Данная строка представляет собой родовое
  -- об'явление. Это - об'явление родового
  -- параметра. Здесь NUM - родовый параметр
  -- типа, который согласовывается с любым
  -- фиксированным типом. Любой фактический
  -- параметр, представляющий собой имя фик-
  -- сированного типа, может быть согласован
  -- с NUM.
  package FIXED_IO is
  -- Содержимое этой части приведено в
  -- приложении В.
end FIXED_IO;
```

В заключение приведем средства для перечисляемых типов:

```
generic
  type ENUM is (<>) ;
  -- Данная строка представляет собой родовое
  -- об'явление. Это - об'явление родового
  -- параметра. Здесь ENUM - родовый параметр
  -- типа, который согласовывается с любым
  -- перечисляемым типом. Любой фактический
  -- параметр, представляющий собой имя пере-
  -- числяемого типа, может быть согласован
  -- с ENUM.
  package ENUMERATION_IO is
  -- Содержимое этой части приведено в
  -- приложении В.
end ENUMERATION_IO;
```

В приложении В есть и другие родовые пакеты, такие, как DIRECT\_IO (Прямой ввод-вывод). Кратко опишем спецификацию родового объявления для DIRECT\_IO :

```
generic
  type ELEMENT_TYPE is private;
  -- Данная строка представляет собой родовое
  -- об'явление. Это - об'явление родового
  -- параметра. Здесь ELEMENT_TYPE - родовый
  -- параметр типа, который согласовывается
```



```

-- с любым типом, допускающим присва-
-- ивание и проверку на равенство и нера-
-- венство. Поэтому фактический параметр
-- может являться именем частного типа,
-- именем комбинированного типа, именем
-- регулярного типа, именем целого типа,
-- именем перечисляемого типа, именем
-- плавающего типа, а также именем неко-
-- торых других типов.
package DIRECT_IO is
-- Содержимое этой части приведено в
-- приложении В.
end DIRECT_IO;
```

**Пример.** Продемонстрируем гибкость, обеспечиваемую родовыми параметрами частного типа, на примере конкретизаций пакета `DIRECT_IO` с использованием ряда типов в качестве фактических параметров:

```

with SECURITIES, DIRECT_IO;
use SECURITIES;
-- Здесь предполагается, что пакет SECURITIES
-- (см. разд. 7.2) уже оттранслирован.
package INSTANT_DIR_IO is
--
package DIR_RATE is new DIRECT_IO ( DATE );
package DIR_SECURITY is new
    DIRECT_IO ( SECURITY );
package DIR_INTEREST is new
    DIRECT_IO ( INTEREST );
package DIR_INT_KIND is new
    DIRECT_IO ( INTEREST_KIND );
--
end INSTANT_DIR_IO;
```

Пакет `INSTANT_DIR_IO` — это совокупность из четырех пакетов, каждый из которых способен целиком использовать ресурсы, предоставляемые родовым пакетом `DIRECT_IO` (процедуры, функции и типы). Родовой параметр частного типа замещается каждым из четырех фактических параметров, создавая четыре независимых пакета — каждый со своим набором типов, функций, процедур и т. д.

Обратите внимание на то, что родовое объявление частного типа или ограниченного частного типа имеет довольно-таки общий вид и предоставляет пользователю значительную свободу в отношении выбора типа фактического параметра.

При объявлении в пакете частных и ограниченных частных типов на использование объектов этих типов накладываются очень жесткие ограничения. Однако если они используются как родовые параметры, то становится верным обратное утверждение: в качестве фактических параметров можно использовать любые типы независимо от того, какие ограничения (такие же или меньшие) накладываются на допустимые для этих типов операции.

Пакет `SEQUENTIAL_IO` из приложения В — это еще один пример родового пакета. Его родовое объявление такое же, как и у пакета `DIRECT_IO`. В следующей главе будет изучен каждый из пакетов, входящих в приложение В. На примерах будет показано их применение.

Приложение В, помимо указанных выше родовых пакетов, содержит также спецификацию пакета `STANDARD`. Этот пакет имеет в своем составе предопределенные объекты (такие, как константы, обозначающие специальные символы кода ASCII), типы (`BOOLEAN`, `INTEGER`, `FLOAT` и т. д.) и функции ("=" для целых чисел, строк и

т. д.), а также фразы представления. Этот пакет непосредственно видим в любой программе на Аде. Он играет особую роль для тех программ, которые будут детально рассмотрены в гл. 9.

Может создаться впечатление, что применение родовых средств аналогично простой подстановке текста в программу. Однако это верно лишь отчасти, так как после замены формальных родовых параметров фактическими параметрами производится обработка конкретизированного пакета или подпрограммы непосредственно в той точке программы, где они появляются. Подстановка текста, напротив, применяется к уже обработанным ресурсам.

Составление родовых пакетов — весьма сложная задача. Разъяснение методов проектирования и составления родовых пакетов выходит за рамки настоящей книги. Вместо этих разъяснений здесь были приведены примеры достаточно прямолинейного использования родовых пакетов и их конкретизации.

## УПРАЖНЕНИЯ

1. Перепишите заново программу `DATE_CONVERSION` из гл. 2, используя последнюю версию пакета `CHECK_DATES_ALT` из разд. 7.2.
2. Перепишите заново программу `NAME_PHONE` из гл. 4, используя пакет `LINE_HANDLING` из разд. 7.2.
3. Выделите из программы `GR_POINT_AVE` (см. гл. 4) нужную спецификацию пакета и тело пакета для работы со студенческими ведомостями успеваемости. Перепишите эту программу заново с использованием нового пакета.
4. Увеличьте возможности пакета `LINE_HANDLING` (Обработка строк) путем составления спецификаций и тел новых подпрограмм

Функция	Описание
EXTRACT	Выделяет часть строки <code>WRK_LINE</code> (т. е. подстроку), имеющую заданную длину и начинающуюся с нужной позиции исходной строки
APPEND	Имеет два формальных параметра типа <code>T_REC</code> и вырабатывает значение типа <code>T_REC</code> , компонента <code>WRK_LINE</code> которого является сцеплением компонент <code>WRK_LINE</code> этих двух формальных параметров
LOCATE	Имеет два формальных параметра типа <code>T_REC</code> , обозначаемых как <code>F_REC_1</code> и <code>F_REC_2</code> . Значение, вырабатываемое функцией, указывает начальную позицию первого вхождения компоненты <code>F_REC_1.WRK_LINE</code> в <code>F_REC_2.WRK_LINE</code>

5. Напишите пакет (спецификацию и тело), содержащий две функции. Одна из них преобразует целые числа в строки символов, а другая — строки символов в целые числа.

# Пакеты ввода-вывода в языке Ада

## 8.1. ВВЕДЕНИЕ В ПАКЕТЫ ВВОДА-ВЫВОДА

### 8.1.1. Концепция файлов

В Аде имеется ряд пакетов, предназначенных для работы с файлами. *Файлы* (имеются в виду внешние файлы) — это совокупности элементов данных, размещенные за пределами программы, обычно на внешнем устройстве, таком, как магнитный диск или лента. Пока будем предполагать, что данные в файле принадлежат к одному и тому же типу<sup>1)</sup>. Во внешнем файле, если только он не пуст, всегда существует первый по порядку элемент. С другой стороны, считается, что файл *неограничен*, так как теоретически после последнего элемента всегда можно добавить еще один. Так, после элемента с номером 10001 можно добавить элемент с номером 10002. На практике же размеры внешних файлов всегда лимитированы реальным объемом памяти внешнего устройства или особенностями конкретной реализации системы.

Может создаться впечатление, что внешние файлы подобны массивам. Однако существуют три важных различия. Во-первых, массив составляет часть программы, а внешний файл является частью ее операционного окружения. Во-вторых, границы массива известны во время выполнения программы, а верхняя граница внешнего файла, как пояснялось выше, может быть неизвестна. В-третьих, доступ к элементам массива производится единообразным способом<sup>2)</sup>, а доступ к элементам файла может выполняться различными методами.

### 8.1.2. Пакеты языка Ада, предназначенные для работы с файлами

В Аде есть ряд пакетов, реализующих основные операции ввода-вывода. Эти пакеты обеспечивают обмен информацией между программами и внешними файлами. В Аде существуют следующие пакеты ввода-вывода: `DIRECT_IO` (Прямой ввод-вывод, т.е. работа с файлами прямого доступа), `SEQUENTIAL_IO` (Последовательный ввод-вывод), `TEXT_IO` (Текстовый ввод-вывод) и `LOW_LEVEL_IO` (Ввод-вывод при помощи средств низкого уровня). Предполагается, что пакет `LOW_LEVEL_IO` обслуживает не слишком часто встречающиеся на практике устройства<sup>3)</sup>. Поэтому в настоящей книге он не рассматривается. Спецификации остальных пакетов ввода-вывода приведены в приложении В.

Беглый взгляд на остальные три пакета показывает, что в них входят некоторые типы и подпрограммы, имеющие одинаковые имена. Во всех этих пакетах имеются ограниченный приватный тип `FILE_TYPE`, перечисляемый тип `FILE_MODE` и процедуры с именами `OPEN`, `CLOSE`, `CREATE` и `DELETE`. Эти процедуры будут описаны в следующих разделах.

<sup>1)</sup> В общем случае это, разумеется, неверно. — *Прим. перев.*

<sup>2)</sup> С помощью индексов. — *Прим. перев.*

<sup>3)</sup> Ввод-вывод низкого уровня позволяет учесть все особенности работы конкретных устройств и как следствие повысить эффективность операций ввода-вывода. Поэтому он пригоден не только для нестандартных устройств. — *Прим. перев.*

### 8.1.3. Присваивание имени файлу

При создании файлов им даются некоторые имена. Имя (NAME) файла принадлежит к типу STRING и служит формальным параметром для ряда подпрограмм из пакетов ввода-вывода. Оно однозначно идентифицирует файл во внешней среде. Файл известен в операционном окружении языка Ада, которое включает операционные системы и прочие языки, под некоторым обозначением, которое является фактическим параметром, соответствующим NAME. В области действия имен программы это *внешнее имя* получает новое обозначение. Новое имя действует до тех пор, пока файл используется, и оно сродни *внутреннему имени*. Это — значение фактического параметра, соответствующего формальному параметру FILE типа FILE\_TYPE.

Тип FILE\_TYPE используется для внутренней идентификации внешнего файла. Всякий раз, когда в книге будет использоваться термин *файл* (в противоположность термину *внешний файл*), будет иметься в виду формальный объект типа FILE\_TYPE.

Объявление типа FILE\_MODE (Режим обмена информацией с файлом) существует в двух разновидностях. Первая версия объявления

```
type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
```

относится к пакету DIRECT\_IO, а вторая

```
type FILE_MODE is (IN_FILE, OUT_FILE);
```

— к пакетам SEQUENTIAL\_IO и TEXT\_IO.

Параметр типа FILE\_MODE определяет направление передачи информации между внешним файлом и программой. Если объект типа FILE\_MODE принимает значение IN\_FILE, то разрешается чтение данных, т. е. только передача информации из файла в программу. Для значения параметра, равного OUT\_FILE, данные записываются во внешний файл. А если режим обмена информацией с файлом устанавливается равным INOUT\_FILE, то можно выполнять как чтение данных из файла, так и запись в него. Этот режим допускается только при использовании пакета DIRECT\_IO<sup>1)</sup>.

### 8.1.4. Создание файла

Если внешний файл не существует, то он должен быть создан при помощи обращения к процедуре CREATE. Объявление этой процедуры одинаково для всех пакетов ввода-вывода и отличается лишь значением параметра MODE, принимаемым по умолчанию. Объявление имеет вид

```
procedure CREATE (FILE : in out FILE_TYPE ;
  MODE : in FILE_MODE := INOUT_FILE ;
  NAME : in STRING := "" ;
  FORM : in STRING := "" );
```

Эта процедура входит в состав пакета DIRECT\_IO. В пакетах SEQUENTIAL\_IO и TEXT\_IO умалчиваемое значение формального параметра MODE равно OUT\_FILE.

В качестве значения параметра NAME, принимаемого по умолчанию, берется пустая строка. Если для NAME не задан фактический параметр или же если он равен пустой строке, то создается так называемый временный файл, доступ к которому становится невозможным после завершения главной программы. Если же фактический параметр, соответствующий NAME, указывается, то он должен представлять собой внешнее имя файла и обязан подчиняться правилам обозначения внешних файлов,

<sup>1)</sup> Точнее, конкретизации этого родового пакета. — Прим. перев.

принятым на конкретной ЭВМ. Очевидно, что параметр NAME зависит от реализации языка.

Четвертый формальный параметр процедуры CREATE—это строка FORM. Данный формальный параметр также является зависящим от реализации. Он может определять различные характеристики внешнего файла, такие, как коэффициент блокирования, период сохранения файла и т. п. Если этот параметр опущен, то в силу вступят действующие на данном ЭВМ умалчиваемые значения.

**Пример вызова процедуры CREATE:**

```
CREATE( FILE => TRANS_FILE,
      -- Этот файл может быть рабочим
      -- файлом, т.е. мы можем записывать
      -- в него вспомогательную информацию.
      MODE => INOUT_FILE,
      -- Этот режим допустим только для
      -- файлов с прямой организацией.
      NAME => "FY8603.DAT",
      -- В этой строке задается внешнее имя
      -- файла. Здесь оно обозначает третий
      -- месяц 1986 : финансового года.
      FORM => " " ) ;
      -- Для этого параметра будет взято
      -- значение, принятое по умолчанию.
      -- Вот пример этого значения для языка
      -- Ада, реализованного на ЭВМ типа VAX:
      -- FORM => "ORGANIZATION SEQUENTIAL";
```

Здесь предполагается, что конкретизация пакета DIRECT\_IO видима непосредственно. Пользуясь позиционной формой записи, можно записать эквивалентное обращение к этой процедуре

```
CREATE (TRANS_FILE, INOUT_FILE, "FY8603.DAT", " " );
```

### 8.1.5. Открытие файла

Если процедура создания закончилась успешно, т.е. не возникло исключительной ситуации, то созданный файл будет находиться в открытом состоянии. Открытие файла означает установление связи между созданным внешним файлом и объектом типа FILE\_TYPE. (Вспомните, что под термином *файл* понимается объект типа FILE\_TYPE.)

Внешний файл мог существовать и до начала выполнения программы, т.е. он мог быть создан раньше. Тогда связь между объектом типа FILE\_TYPE и внешним файлом будет установлена после успешного завершения процедуры OPEN. Критерием *успешности* здесь служит отсутствие каких-либо исключительных ситуаций во время выполнения процедуры OPEN. Процедура CLOSE прерывает связь между объектом типа FILE\_TYPE, т.е. собственно файлом, и внешним файлом.

В объявлении процедуры OPEN присутствуют те же самые формальные параметры, что и у процедуры CREATE:

```
procedure OPEN ( FILE : in out FILE_TYPE ;
                 MODE : in   FILE_MODE ;
                 NAME : in   STRING   ;
                 FORM : in   STRING   := " " );
```

В отличие от процедуры CREATE здесь нет умалчиваемых значений для параметров MODE и NAME, а это означает, что задание параметров MODE и NAME является обязательным. Поэтому при открытии файла всегда следует указывать режим обмена информацией с ним, т.е. параметр MODE.

**Пример** открытия файла:

```
OPEN ( FILE => TRANS_FILE,
        MODE => INOUT_FILE,
        -- Этот режим подразумевает использо-
        -- вание пакета DIRECT_IO.
        NAME => "FY8603.DAT",
        -- Здесь задается внешнее имя файла.
        FORM => "" );
        -- Здесь принимается значение, выбира-
        -- емое по умолчанию.
```

В позиционной форме вызов процедуры OPEN можно записать так:

```
OPEN (TRANS_FILE, INOUT_FILE, "FY8603.DAT", "" );
```

Здесь опять-таки предполагается, что конкретизация пакета DIRECT\_IO видима непосредственно.

### 8.1.6. Неизменяемые характеристики файла

Внешнему файлу (после его создания) присваиваются некоторые *неизменяемые характеристики*. Точный вид этих характеристик зависит от конкретной системы, однако следует ожидать, что это будут параметры, характеризующие организацию внешнего файла и, возможно, размер записей в файле, а также права доступа к нему. При открытии этого внешнего файла в будущем данные неизменяемые характеристики сохранятся. Если при открытии файла процедурой OPEN указываемые для нее параметры не будут согласоваться с неизменяемыми характеристиками, то возникнут исключительные ситуации. Это может, например, случиться, если вид организации файла, указанный в параметре FORM процедуры OPEN, окажется несовместимым с организацией внешнего файла, установленной при его создании. Исключительные ситуации могут возникать и во многих других случаях, например если предпринимается попытка повторно открыть уже открытый файл. В пакете IO\_EXCEPTIONS определена большая часть исключительных ситуаций, связанных с обработкой файлов. Эти исключительные ситуации и их обработка будут рассмотрены в гл. 11.

### 8.1.7. Заккрытие файла

Объявление процедуры CLOSE таково:

```
procedure CLOSE (FILE : in out FILE_TYPE);
```

**Пример** вызова этой процедуры:

```
CLOSE (TRANS_FILE);
```

Во время выполнения программы внешний файл можно открывать и закрывать неоднократно. Собственно файл (внутренний объект) также можно несколько раз открывать и закрывать во время выполнения программы. При этом можно один и тот же файл (внутренний объект) при открытии связывать в разное время с разными внешними файлами.

### 8.1.8. Уничтожение внешнего файла

Если внешний файл открыт, то его можно физически уничтожить с помощью обращения к процедуре DELETE. Ее объявление таково:

```
procedure DELETE (FILE : in out FILE_TYPE);
```

Пример вызова этой процедуры:

```
DELETE (TRANS_FYLE);
```

### 8.1.9. Пример программы

Следующая программа иллюстрирует описанные выше средства Ады. Она создает при помощи родового пакета DIRECT\_IO 12 внешних файлов с внешними именами от FY8601.DAT до FY8612.DAT.

#### Программа CREATE\_12\_TRANS\_FILES

```
with DIRECT_IO ;
PROCEDURE CREATE_12_TRANS_FILES is
-- Как отмечалось в конце гл.7, пакеты
-- DIRECT_IO и SEQUENTIAL_IO имеют родовой
-- параметр ELEMENT_TYPE.
type TRANS_RECORD is
  record
    INFO : STRING ( 1 .. 80 ) ;
  end record ;
package DIR_IO is new DIRECT_IO (
  ELEMENT_TYPE => TRANS_RECORD ) ;
use DIR_IO
TRANS_FILE : FILE_TYPE ;
-- Мы конкретизировали пакет DIRECT_IO
-- для типа элементов TRANS_RECORD
EXT_NAME : STRING ( 1 .. 10 ) := "FY8601.DAT" ;
begin
  for I in 1 .. 12
  loop
    CREATE ( FILE => TRANS_FILE,
              MODE => INOUT_FILE,
              NAME => EXT_NAME,
              FORM => "" ) ;
    CLOSE ( TRANS_FILE ) ;
    if EXT_NAME (6) /= '9'
    then
      EXT_NAME := EXT_NAME ( 1 .. 5 ) &
        CHARACTER'SUCC( EXT_NAME(6) ) &
        EXT_NAME ( 7 .. 10 ) ;
    else
      EXT_NAME := EXT_NAME ( 1 .. 4 ) &
        "10" & EXT_NAME ( 7 .. 10 ) ;
      -- Внешнее имя образуется путем
      -- увеличения на единицу послед-
      -- ней цифры, стоящей перед де-
      -- сятичной точкой, если только
      -- цифра не равна девяти. Обра-
      -- тите внимание, что здесь со-
      -- здаются 12 внешних файлов,
```

```

-- но при этом используется то-
-- лько один внутренний файл,
-- который периодически закрыв-
-- ается и повторно (с помощью
-- процедуры CREATE) открывается.

end if;
end loop;
end CREATE_12_TRANS_FILES;

```

### 8.1.10. Функции, общие для всех пакетов ввода-вывода

В Аде есть функции, общие для всех пакетов ввода-вывода. Они позволяют проверить состояние, в котором находится файл. Приведем их объявления.

```
function NAME (FILE : FILE_TYPE) return STRING;
```

Результатом выполнения этой функции является внешнее имя файла, указанного в качестве аргумента.

```
function END_OF_FILE (FILE : FILE_TYPE) return BOOLEAN;
```

Значение данной функции будет равно TRUE, если в файле больше не осталось элементов (т.е. это — конец файла). Другими словами, если функция будет вызвана после считывания последнего элемента файла, то она выработает значение TRUE.

```
function IS_OPEN (FILE : FILE_TYPE) return BOOLEAN;
```

Значение этой функции будет равно TRUE, если заданный файл открыт.

```
function MODE (FILE : FILE_TYPE) return FILE_MODE;
```

Данная функция выдает текущий режим обмена информацией с представленным файлом.

```
function FORM (FILE : FILE_TYPE) return STRING;
```

Значением этой функции будет строка, содержащая системно-зависимые параметры для указанного файла.

## 8.2. ОБРАБОТКА ПОСЛЕДОВАТЕЛЬНЫХ ФАЙЛОВ

*Последовательные файлы* можно рассматривать как последовательность элементов, принадлежащих к типу `ELEMENT_TYPE`, который является родовым формальным параметром пакета `SEQUENTIAL_IO`. Последовательный порядок расположения элементов в файле обуславливает аналогичный порядок обработки этих элементов. После открытия файла становится доступным для чтения его первый элемент. Если же нужно получить значение третьего элемента файла, то перед этим придется прочитать первые два элемента.

Операции, требующиеся для обработки последовательного файла, реализуются родовым пакетом `SEQUENTIAL_IO`, спецификация которого приведена в приложении В. Вот объявление процедуры последовательного чтения `READ`:

```
procedure READ (FILE : in FILE_TYPE;
                ITEM : out ELEMENT_TYPE);
```

Эта процедура считывает из заданного файла один элемент и возвращает его значение через параметр `ITEM`. При этом предполагается, что файл открыт и режим обмена информацией для него — `IN_FILE` (входной). При следующем вызове процедура считывает



следующий элемент. Если элемент будет последним в файле, то при последующем вызове функции `END_OF_FILE` ею будет выработываться значение `TRUE`.

Вот объявление процедуры последовательной записи `WRITE`:

```
procedure WRITE (FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE);
```

Эта процедура записывает в файл (после последнего элемента) значение нового элемента. При этом предполагается, что файл открыт и режим обмена информацией для него — `OUT_FILE` (выходной).

В некоторых случаях требуется многократная обработка одного и того же файла. При этом можно воспользоваться процедурой `RESET`, объявление которой помещено ниже. Ее можно использовать вместо того, чтобы многократно закрывать и повторно открывать один и тот же файл.

```
procedure RESET (FILE : in FILE_TYPE;
                 MODE : in FILE_MODE);
```

Эта процедура делает уже открытый файл доступным для чтения или записи, начиная с первого элемента файла. При закрытии и повторном открытии файла получится точно такой же результат. Однако процедура `RESET` не разрывает связи между внешним файлом и внутренним файлом. Поэтому ее применение экономит машинное время.

Процедура

```
procedure RESET (FILE : in FILE_TYPE);
```

аналогична предыдущей. Однако при использовании этой версии процедуры `RESET` нельзя изменить параметр `MODE`.

### 8.2.1. Построение последовательного файла

Здесь представлена программа на Аде, в которой использован родовой пакет `SEQUENTIAL_IO`. Входные и выходные данные этой программы совпадают с данными программы `RECUR_PROC_GRADES` из гл. 5, являющейся в свою очередь модификацией программы `ACCESS_GRADES` из гл. 3. Примеры входных данных показаны на рис. 3.7 и рис. 5.1. Однако в данной версии программы введено дополнительное ограничение: ключи к ответам отсортированы в соответствии с алфавитным порядком наименований предметов, а студенческие ответы отсортированы по личным номерам студентов. Кроме того, для строк с одинаковыми номерами студентов заранее выполнена сортировка по наименованиям предметов.

Программа, вдобавок, строит последовательный файл, содержащий информацию о контрольных работах, и еще один последовательный файл, вмещающий ответы студентов.

#### Программа `SEQ_PROC_GRADES`

```
with TEXT_IO; use TEXT_IO;
with SEQUENTIAL_IO;
procedure SEQ_PROC_GRADES is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type CHOICES is range 1 .. 5;
  type POSSIBLE_QUESTIONS is range 20 .. 50;
  package CHO_IO is new INTEGER_IO(CHOICES);
  use CHO_IO;
  package POSS_IO is new
    INTEGER_IO(POSSIBLE_QUESTIONS);
```

```

use POSS_IO;
DNO_QUESTIONS : POSSIBLE_QUESTIONS;
type ANSWERS is array ( 1 .. DNO_QUESTIONS ) of
    CHOICES;
type TEST_KEY is
    record
        SUBJ      : STRING(1 .. 5);
        NO_QUESTIONS : POSSIBLE_QUESTIONS;
        KEY_ANSWERS : ANSWERS;
    end record;
CURR_TEST : TEST_KEY;
GOOD_ANSWERS : INTEGER ;
type IN_REC is
    record;
        STUDENT_ID : STRING ( 1 .. 10 );
        SUBJECT_ID : STRING ( 1 .. 5 );
        STUDENT_ANSWERS : ANSWERS;
    end record;
CURR_REC : IN_REC;
type T_PAIR is
    record
        SUBJ_MAT : STRING ( 1 .. 5 );
        SCORE    : NATURAL;
    end record;
CURR_PAIR : T_PAIR;
type SEMESTER_TESTS is array (1 .. 25 )
    of T_PAIR;
type BIG_REC is
    record
        BIG_ST_ID : STRING ( 1 .. 10 );
        NO_TESTS  : NATURAL;
        ST_TESTS  : SEMESTER_TESTS;
    end record;
-- Идентификаторы, использованные в тексте,
-- совпадают с идентификаторами из програм-
-- мы RECUR_PROC_GRADES и имеют тот же самый
-- смысл, за исключением того, что здесь не
-- употребляются ссылочные переменные.
CURR_OUT_REC : BIG_REC;
package TEST_IO is new SEQUENTIAL_IO(TEST_KEY);
use TEST_IO;
package STUD_IO is new SEQUENTIAL_IO(BIG_REC);
use STUD_IO;
-- Две конкретизации пакета SEQUENTIAL_IO необ-
-- ходимы для того, чтобы работать с двумя по-
-- следовательными файлами (файл с названиями
-- контрольных работ и файл со сведениями о
-- студентах).
TEST_FILE : TEST_IO.FILE_TYPE;
STUDENT_FILE : STUD_IO.FILE_TYPE;
procedure SAME_STUD_PROC is
    -- Эта процедура обновляет сведения, хранящи-
    -- еся в структуре CURR_OUT_REC.
begin
    GET(CURR_REC.SUBJECT_ID);
    while not END_OF_FILE(FILE => TEST_FILE) and
        CURR_TEST.SUBJ < CURR_REC.SUBJECT_ID

```

```

loop
  -- Поиск предмета, по которому проводится
  -- контрольная работа, в TEST_FILE.
  READ(FILE => TEST_FILE,
        ITEM => CURR_TEST);
end loop;
if CURR_TEST.SUBJ = CURR_REC.SUBJECT_ID
then
  -- Если условие истинно, то предмет
  -- найден. В противном случае такого
  -- предмета нет в списке.
  GOOD_ANSWERS := 0;
  for J in 1 .. CURR_TEST.NO_QUESTIONS
  loop
    GET(CURR_REC.STUDENT_ANSWERS(J), 1);
    if CURR_REC.STUDENT_ANSWERS(J) =
      CURR_TEST.KEY_ANSWERS(J)
    then
      GOOD_ANSWERS := GOOD_ANSWERS + 1;
    end if;
  end loop;
  NEW_LINE;
  -- Оценка за контрольные работы вычислена.
  -- Теперь следует обновить сведения об ус-
  -- пехе студента.
  CURR_OUT_REC.NO_TESTS :=
    CURR_OUT_REC.NO_TESTS + 1;
  CURR_PAIR.SUBJ_MAT := CURR_REC.SUBJECT_ID;
  CURR_PAIR.SCORE := GOOD_ANSWERS;
  CURR_OUT_REC.ST_TESTS(CURR_OUT_REC.NO_TESTS)
    := CURR_PAIR;
else
  PUT(" No such subject ");
  PUT(CURR_REC.SUBJECT_ID);
end if;
end SAME_STUD_PROC;

procedure NEW_STUD_PROC is
  -- Значение переменной CURR_OUT_REC теперь
  -- получено. Оно отражает результаты преды-
  -- дущих контрольных работ.
begin
  STUD_IO.WRITE(FILE => STUDENT_FILE,
                ITEM => CURR_OUT_REC);
  CURR_OUT_REC.BIG_ST_ID :=
    CURR_REC.STUDENT_ID;
  CURR_OUT_REC.NO_TESTS := 0;
  RESET( FILE => TEST_FILE);
  SAME_STUD_PROC;
end NEW_STUD_PROC;

--
begin
  -- Создать файл контрольных работ.
  TEST_IO.CREATE( FILE => TEST_FILE,
                 MODE => OUT_FILE,
                 -- Необходимо записать в файл ключи ответов
                 -- для контрольных работ.
                 NAME => "TEST_FILE.DAT",
                 FORM => "");

```

```

-- Здесь используются характеристики файла,
-- принятые по умолчанию. Файл TEST_FILE
-- открывается, и в него можно записать
-- первое значение, принадлежащее к типу
-- TEST_KEY.
GET(CURR_TEST.SUBJ);
while CURR_TEST.SUBJ /= "XXXXX"
loop
  GET(CURR_TEST.NO_QUESTIONS, 2);
  DNO_QUESTIONS := CURR_TEST.NO_QUESTIONS;
  SKIP_LINE;
  for I in 1 .. CURR_TEST.NO_QUESTIONS;
  loop
    GET(CURR_TEST.KEY_ANSWERS(I), 1);
  end loop;
  TEST_IO.WRITE(FILE => TEST_FILE,
    ITEM => CURR_TEST);
  SKIP_LINE;
  GET(CURR_TEST.SUBJ);
end loop;
-- Файл контрольных работ теперь построен.
-- Файл нужно читать последовательно, поэтому
-- надо его закрыть, а затем открыть повторно
-- с режимом IN_FILE (ВХОДНОЙ). Вместо этого
-- можно было бы воспользоваться процедурой
-- RESET.
TEST_IO.CLOSE ( FILE => TEST_FILE );
TEST_IO.OPEN ( FILE => TEST_FILE,
  MODE => IN_FILE,
  -- Нужно считывать из файла ключи ответов.
  NAME => "TEST_FILE.DAT",
  FORM => "" );
-- Этот файл будет устанавливаться заново в
-- исходное положение для каждого нового
-- личного номера студента с помощью проце-
-- дуры RESET. Вместо этого его можно закры-
-- вать и открывать повторно.
STUD_IO.CREATE( FILE => STUDENT_FILE,
  MODE => OUT_FILE,
  -- Здесь требуется записывать сведения об
  -- успеваемости студентов.
  NAME => "STUDENT_FILE.DAT"),
  FORM => "" );
-- Здесь используются характеристики файла,
-- принятые по умолчанию.
SKIP_LINE;
GET(CURR_REC.STUDENT_ID);
CURR_OUT_REC.BIG_ST_ID := CURR_REC.STUDENT_ID;
CURR_OUT_REC.NO_TESTS := 0;
while CURR_REC.STUDENT_ID /= "9999999999"
loop
  if CURR_OUT_REC.BIG_ST_ID /=
    CURR_REC.STUDENT_ID
  then
    NEW_STUD_PROC;
  else
    SAME_STUD_PROC;
  end if;

```

```

    GET (CURR_REC.STUDENT_ID);
end loop;
-- Обработаны все строки входных данных со
-- сведениями о студентах. Однако послед-
-- няя структура со сведениями об успеха-
-- мости студентов еще не записана.
NEW_STUD_PROC;
-- Теперь структура со сведениями о последнем
-- студенте записана.
STUD_IO.RESET( FILE => STUDENT_FILE,
-- Файл со сведениями о студентах установ-
-- ливается на начало. Режим работы с фай-
-- лом изменяется так, чтобы можно было
-- считывать данные из него.
MODE => INFILE );
while not STUD_IO.END_OF_FILE
    (FILE => STUDENT_FILE)
-- Условие будет истинным, если при
-- следующей попытке чтения значения
-- переменной из файла STUDENT_FILE
-- данные окажутся исчерпанными.
loop
STUD_IO.READ(STUDENT_FILE, CURR_OUT_REC);
-- Этот оператор выполняет считывание сле-
-- дующего значения из файла STUDENT_FILE.
PUT(CURR_OUT_REC.BIG_ST_ID);
for I in 1 .. CURR_OUT_REC.NO_TESTS
loop
PUT(CURR_OUT_REC.ST_TESTS(I).SUBJ_MAT);
PUT(CURR_OUT_REC.ST_TESTS(I).SCORE );
end loop;
end loop;
TEST_IO.CLOSE(TEST_FILE);
TEST_IO.CLOSE(STUDENT_FILE);
end SEQ_PROC_GRADES;

```

### 8.2.2. Объединение двух файлов

Далее представлен пример объединения двух внешних файлов, имеющих внешние имена `STUDENT_FILE_1.IN.DAT` и `STUDENT_FILE_2.IN.DAT`. Предполагается, что на вход поступают два отсортированных файла. В нашем случае два внешних файла содержат элементы типа `BIG_REC` и отсортированы по значениям компонент `BIG_ST_ID` этих элементов. На выходе программы объединения должен быть образован третий файл, элементы которого принадлежат к тому же типу, что и элементы входных файлов. В этом файле будет содержаться объединенная информация, считанная из входных файлов.

Сделаем следующее упрощение: если два элемента из разных файлов имеют идентичные значения компонент `BIG_ST_ID`, а учебные предметы, по которым выполняются контрольные работы, отличаются, то общее число этих предметов не превышает 25, а наименования предметов из файла `STUDENT_FILE_1.IN.DAT` предшествуют<sup>1)</sup> названиям предметов из файла `STUDENT_FILE_2.IN.DAT`. В упражнениях, приведенных в конце данной главы, эти ограничения будут несколько ослаблены. На рис. 8.1 показан пример входных файлов и выходного файла, образованного в результате их объединения.

<sup>1)</sup> В соответствии с алфавитным порядком.—Прим. перев.

Файл STUD\_IN\_1  
CURR\_REC\_1\_IN

BIG_ST_ID	NO_TESTS	ST_TESTS				
		(1)	(2)	(3)	(4) ..	(25)
1234567890	3	AST01 75	CMP05 88	ENG01 89		
2222222222	2	CMP01 89	PHY03 77			
3333333333	2	BSN01 75	CMP01 89			

Файл STUD\_IN\_2  
CURR\_REC\_2\_IN

BIG_ST_ID	NO_TESTS	ST_TESTS				
BIG_ST_ID	NO_TESTS	ST_TESTS				
		(1)	(2)	(3)	(4) ..	(25)
1234567890	1	PHY01 55				
3333333333	2	ENG01 75	PHY01 95			

Файл STUD\_OUT\_FILE  
CURR\_REC\_OUT

BIG_ST_ID	NO_TESTS	ST_TESTS				
		(1)	(2)	(3)	(4) ..	(25)
1234567890	4	AST01 75	CMP05 88	ENG01 89	PHY01 55	
2222222222	2	CMP01 89	PHY03 77			
3333333333	4	BSN01 75	CMP01 89	ENG01 75	PHY01 95	

Рис. 8.1. Примеры файлов для программы MERGE\_PROC\_GRADES.

### Программа MERGE\_PROC\_GRADES

```
with TEXT_IO; use TEXT_IO;
with SEQUENTIAL_IO;
procedure MERGE_PROC_GRADES is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type T_PAIR is
    record
      SUBJ_MAT : STRING ( 1 .. 5 );
      SCORE    : NATURAL;
    end record;
  CURR_PAIR : T_PAIR;
  type SEMESTER_TESTS is array (1 .. 25 )
    of T_PAIR;
  type BIG_REC is
    record
      BIG_ST_ID : STRING ( 1 .. 10 );
      NO_TESTS  : NATURAL;
      ST_TESTS  : SEMESTER_TESTS;
    end record;
  -- Идентификаторы, использованные в тексте,
  -- до сих пор совпадают с идентификаторами из
  -- программы RECUR_PROC_GRADES и имеют тот же
  -- смысл.
  package STUD_IO is new SEQUENTIAL_IO(BIG_REC);
  use STUD_IO;
  CURR_REC_1_IN, CURR_REC_2_IN,
  CURR_REC_OUT : STUD_IO.BIG_REC;
```

```

STUD_IN_1, STUD_IN_2, STUD_OUT_FILE :
                                STUD_IO.FILE_TYPE;
--
procedure READ_1 is
begin
    READ(FILE => STUD_IN_1,
         ITEM => CURR_REC_1_IN);
end READ1;
--
procedure READ_2 is
begin
    READ(FILE => STUD_IN_2,
         ITEM => CURR_REC_2_IN);
end READ2;
--
procedure WRITE_OUT is
begin
    WRITE(FILE => STUD_OUT_FILE,
          ITEM => CURR_REC_OUT );
end WRITE_OUT;
--
procedure COPY_1 is
begin
    while not STUD_IO.END_OF_FILE(STUD_IN_1)
    loop
        READ_1;
        CURR_REC_OUT := CURR_REC_1_IN;
        WRITE_OUT;
    end loop;
    PUT(" First input file was last processed");
end COPY_1;
--
procedure COPY_2 is
begin
    while not STUD_IO.END_OF_FILE(STUD_IN_2)
    loop
        READ_2;
        CURR_REC_OUT := CURR_REC_2_IN;
        WRITE_OUT;
    end loop;
    PUT("Second input file was last processed");
end COPY_2;
--
begin
    STUD_IO.OPEN(FILE => STUD_IN_1,
                 MODE => IN_FILE,
                 NAME => "STUDENT_FILE_1.IN.DAT",
                 FORM => " " );
    STUD_IO.OPEN(FILE => STUD_IN_2,
                 MODE => IN_FILE,
                 NAME => "STUDENT_FILE_2.IN.DAT",
                 FORM => " " );
    STUD_IO.CREATE(FILE => STUD_OUT_FILE,
                  MODE => OUT_FILE,
                  NAME => "STUD-MERGED-FILE.DAT",
                  FORM => " " );
    if not STUD_IO.END_OF_FILE(STUD_IN_1) and
       not STUD_IO.END_OF_FILE(STUD_IN_2)

```

```

then
  READ_1;
  READ_2;
loop
  if CURR_REC_1_IN.BIG_ST_ID <
    CURR_REC_2_IN.BIG_ST_ID
  then
    -- Запись данных из первого файла.
    CURR_REC_OUT := CURR_REC_1_IN;
    WRITE_OUT;
    if STUD_IO.END_OF_FILE(STUD_IN_1)
    then
      -- Записан последний элемент из пер-
      -- вого входного файла. Теперь запи-
      -- шем текущий элемент из второго
      -- входного файла. В противном слу-
      -- ае этот элемент будет отброшен в
      -- процедуре COPY_2. Слияние файлов
      -- заканчивается после окончания
      -- выполнения COPY_2.
      CURR_REC_OUT := CURR_REC_2_IN;
      WRITE_OUT;
      COPY_2;
      exit;
    end if;
    READ_1;
  elsif CURR_REC_1_IN.BIG_ST_ID >
    CURR_REC_2_IN.BIG_ST_ID
  then
    -- Теперь выполняется запись данных
    -- из второго файла.
    CURR_REC_OUT := CURR_REC_2_IN;
    WRITE_OUT;
    if STUD_IO.END_OF_FILE(STUD_IN_2)
    then
      -- Записан последний элемент из вто-
      -- рого входного файла. Теперь запи-
      -- шем текущий элемент из первого
      -- входного файла. В противном слу-
      -- чае этот элемент будет отброшен в
      -- процедуре COPY_2. Слияние файлов
      -- заканчивается после окончания
      -- выполнения COPY_1.
      CURR_REC_OUT := CURR_REC_1_IN;
      WRITE_OUT;
      COPY_1;
      exit;
    end if;
    READ_2;
  else
    -- Для равенства здесь складываются количес-
    -- тва контрольных работы у студентов.
    CURR_REC_OUT := CURR_REC_1_IN;
    CURR_REC_OUT.NO_TESTS :=
      CURR_REC_OUT.NO_TESTS +
      CURR_REC_2_IN.NO_TESTS;
    CURR_REC_OUT.ST_TESTS

```



```

        (CURR_REC_1.IN.NO_TESTS + 1 ..
          CURR_REC_OUT.NO_TESTS) :=
CURR_REC_2.IN.ST_TESTS
  (1 .. CURR_REC_2.IN.NO_TESTS);
-- Это - присваивание вырезки.
WRITE_OUT;
if STUD_IO.END_OF_FILE(STUD_IN_2)
then
  COPY_1;
  exit;
end if;
-- После того как достигнут конец одного
-- файла, копируем другой файл.
if STUD_IO.END_OF_FILE(STUD_IN_1)
then
  COPY_2;
  exit;
end if;
READ_1;
READ_2;
end if;
-- Здесь объединяются два непустых файла.
-- По крайней мере, один файл обработан.
-- Теперь скопируем оставшийся.
end loop;
elseif STUD_IO.END_OF_FILE(STUD_IN_1)
then
  COPY_2;
else
  -- Если управление попадает сюда, то
  -- файл STUD_IN_2 должен быть пустым.
  COPY_1;
end if;
PUT(" The merge is done ");
STUD_IO.CLOSE(STUD_IN_1);
STUD_IO.CLOSE(STUD_IN_2);
STUD_IO.CLOSE(STUD_OUT_FILE);
end MERGE_PROC_GRADES;

```

## 8.3. ОБРАБОТКА ФАЙЛОВ ПРЯМОГО ДОСТУПА

### 8.3.1. Особенности файлов прямого доступа

*Файл прямого доступа* определяется как последовательность элементов, принадлежащих к типу `ELEMENT_TYPE`, который является родовым формальным параметром пакета `DIRECT_IO`. (В этом данный пакет не отличается от пакета `SEQUENTIAL_IO`.) Однако в отличие от элементов последовательных файлов, обработка которых реализована пакетом `SEQUENTIAL_IO`, элементы, входящие в файл прямого доступа, можно обрабатывать в произвольном порядке, т. е. вне зависимости от их относительного положения в файле. На практике для чтения или записи элемента в файл прямого доступа необходимо указать, помимо прочих фактических параметров, относительное положение этого значения в файле, которое называется *индексом* элемента файла.

Для каждого открытого файла прямого доступа имеется так называемое текущее значение индекса. *Текущее значение* индекса – это значение индекса, которое будет

использоваться при чтении или записи следующего элемента файла, если только программист не изменил значение индекса явно.

Процедура SET\_INDEX, как и ряд других процедур, позволяет управлять относительным положением считываемого или записываемого элемента. Функция INDEX дает возможность определить текущее положение файла. Объявления этих подпрограмм, как показано в приложении В, имеют вид

```
procedure SET_INDEX (FILE : FILE_TYPE;
                     TO : POSITIVE_COUNT);
```

POSITIVE\_COUNT является подтипом типа COUNT, диапазон значений которого от 1 до системно-зависимого значения COUNT\_LAST. Индекс файла устанавливается равным значению фактического параметра, соответствующего формальному параметру TO.

```
function INDEX (FILE : FILE_TYPE) return POSITIVE_COUNT;
```

Эта функция вырабатывает значение, равное текущему индексу файла.

Подпрограммы SET\_INDEX и INDEX можно использовать при любых режимах обмена информацией (IN\_FILE, OUT\_FILE или INOUT\_FILE) с файлом прямого доступа. Непосредственно после открытия файла текущее значение индекса устанавливается равным 1.

Процедуры READ и WRITE, реализующие операции чтения и записи для файлов прямого доступа, имеют следующие объявления:

```
procedure READ ( FILE : in FILE_TYPE ;
                 ITEM : out ELEMENT_TYPE ;
                 FROM : in POSITIVE_COUNT );
procedure READ ( FILE : in FILE_TYPE ;
                 ITEM : out ELEMENT_TYPE );
```

В первом варианте процедуры READ элемент типа ELEMENT\_TYPE считывается из файла. Относительное положение элемента задается фактическим параметром, соответствующим формальному параметру FROM. Считанное значение будет передано фактическому параметру, соответствующему формальному параметру ITEM. Во втором варианте относительное положение считываемого элемента определяется текущим значением индекса. После выполнения операции ввода процедурой READ текущее значение индекса увеличивается на 1.

При возникновении ошибок могут быть возбуждены различные исключительные ситуации. Например, если предпринимается попытка чтения из файла, который не открыт или имеет режим обмена информацией OUT\_FILE, то возникнут исключительные ситуации. Более подробно они будут рассмотрены в гл. 11.

Объявления процедур вывода WRITE из пакета DIRECT\_IO являются «зеркальным отражением» объявлений процедур READ:

```
procedure WRITE ( FILE : in FILE_TYPE ;
                  ITEM : out ELEMENT_TYPE ;
                  TO : in POSITIVE_COUNT );
procedure WRITE ( FILE : in FILE_TYPE ;
                  ITEM : out ELEMENT_TYPE );
```

В первом варианте будет записано в файл значение фактического параметра, соответствующего формальному параметру ITEM. Относительное положение элемента в файле задается фактическим параметром, соответствующим формальному параметру TO. Во втором варианте относительное положение элемента определяется текущим значением индекса. После завершения выполнения вывода значение индекса увеличивается

ется на единицу. Исключительные ситуации могут возникнуть, если файл не открыт, если режим обмена информацией с файлом – IN\_FILE и в других случаях.

Файлы прямого доступа имеют *размер*, который определяется наибольшим значением индекса, использованным при чтении данных из файла или при записи в него. Текущее значение размера файла вырабатывает функция SIZE, имеющая объявление:

```
function SIZE (FILE : FILE_TYPE) return COUNT;
```

Если текущее значение индекса превысит размер файла прямого доступа, то функция END\_OF\_FILE даст значение TRUE. В противном случае она даст значение FALSE. Объявление этой функции:

```
function END_OF_FILE (FILE : FILE_TYPE) return BOOLEAN;
```

Пакет DIRECT\_IO имеет в своем составе версии процедур RESET, напоминающие одноименные процедуры из пакета SEQUENTIAL\_IO. Их смысл и объявления в этих пакетах совпадают, за исключением того, что параметр MODE может дополнительно принимать значение INOUT\_FILE для файлов прямого доступа.

### 8.3.2. Программа, в которой используется файл прямого доступа

Проиллюстрируем применение родового пакета DIRECT\_IO на примере простой программы, работающей с файлами. Эта программа отслеживает банковские операции для маклера, работающего с иностранной валютой. Файл прямого доступа, внешнее имя которого BANK\_MASTER.DAT, содержит информацию о банках. Предполагается, что файл уже создан и содержит сведения приблизительно о 150 банках.

Этот файл прямого доступа можно обновлять путем добавления данных о новом банке или путем изменения имеющейся информации о банке, однако удаление сведений о банке не допускается. Кроме того, разрешается наводить справки о некоторых банках. Во входных данных программы, поступающих, например, с терминала, задаются вид действия (первым символом строки данных) и, далее, необходимая для выполнения этого действия информация. В последней строке данных ставится код действия 'Z'.

Входные данные имеют следующий формат:

Позиции	Данные
1	Код действия IN_CODE, где 'A' означает добавление сведений, 'C' – их изменение, 'I' – получение справки, а 'Z' – конец работы
2–21	Название банка
22–70	Прочая информация: телефон, адрес, данные о служащих банка, с которыми следует поддерживать контакты и т. д.

Файл прямого доступа содержит элементы типа BANC\_REC. Тип BANC\_REC – это комбинированный тип с вариантной частью. Вариантная часть охватывает три вида структур. Структуры первого вида содержат глобальную информацию, такую, как текущее количество банков в файле и максимально допустимое для файла число банков. Структуры второго вида несут сведения о конкретном банке. И наконец, структуры третьего вида содержат сводные данные, т. е. сведения о номерах сделок с конкретным банком. Сведения о банках отсортированы в алфавитном порядке по названиям банков.

В пакете BANK\_RESOURCES, текст которого приведен ниже, помещаются необходимые ресурсы: объявления, подпрограммы, инициализирующие действия.

```

with DIRECT_IO;
package BANK_RESOURCES is
type BANK_INFO is
  record
    BANK_NAME : STRING(1 .. 20);
    OTHER_INFO : STRING(1 .. 49);
  end record;
type IN_REC is
  record
    IN_CODE : CHARACTER;
    -- Здесь должны появляться символы
    -- A, C, I и Z.
    IN_DATA : BANK_INFO;
  end record;
type REC_KIND is (GLOB_FILE_DATA, MASTER_DATA,
  POSTING_DATA);
type TRANS_NUMBER is
  record
    MO_PART : INTEGER range 1 .. 12;
    DAY_PART : INTEGER range 1 .. 31;
    NO_PART : NATURAL;
  end record;
type POST_INFO is array (1 .. 12)
  of TRANS_NUMBER;
type BANK_REC(M_TYPE : REC_KIND :=
  MASTER_DATA) is
  record
    case M_TYPE is
      when GLOB_FILE_DATA =>
        NO_BANKS : NATURAL;
        -- Эта компонента позволяет подсчитать
        -- общее число банков в файле.
        MAX_NO : NATURAL;
        -- Значение этой переменной опре-
        -- деляет предельно допустимое
        -- количество банков.
        -- Данные о банковских операциях
        -- можно записывать с превышением
        -- этого предела.
      when MASTER_DATA =>
        MAST_HEAD : BANK_INFO;
        FIRST_TRAN : NATURAL;
        -- Эта переменная содержит значение
        -- индекса первой переменной
        -- POSTING_LINE для банка (она равна
        -- нулю, если у данного банка нет опера-
        -- ций).
        LAST_TRAN : NATURAL;
        -- Эта переменная содержит значение
        -- индекса для последней переменной
        -- POSTING_LINE для банка. Она
        -- равна нулю, если банковские
        -- операции отсутствуют.
        LAST_POS : NATURAL range 1 .. 12;
        -- Эта переменная содержит значение
        -- последней позиции в последней
        -- переменной POSTING_LINE для
        -- конкретного банка.
    end case;
  end record;
end package BANK_RESOURCES;

```

```

when POSTING_DATA =>
  POSTING_LINE : POST_INFO;
  NXT_PST_LINE : NATURAL;
  -- Переменная POSTING_LINE содержит
  -- сведения максимально о 12
  -- банковских операциях.
end case;
end record;

```

---

### BANK\_FILE

Первый элемент файла принадлежит к типу BANK\_REC (GLOB\_FILE\_DATA):

```

NO_BANKS    MAX_NO
3            200

```

Элементы файла со второго по четвертый относятся к типу BANK\_REC (MASTER\_DATA):

MAST_HEAD		FIRST_TRAN	LAST_TRAN	LAST_POS
BANK_NAME	OTHER_INFO			
CHASE MANHATTAN	YYYY .. YYY	203	203	2
CHEMICAL BANK	XXXX .. XXX	202	204	3
MARINE MIDLAND	ZZZZ .. ZZZ	201	201	1

Элементы файла с номерами 201–204 имеют тип BANK\_REC (POSTING\_DATA):

POSTING_LINE		NXT_PST_LINE
(1)	(2) ..(12)	
0723005	0723008	0
0723001	0723003 ... 0724002	204
0723002		0
0724005	0724006 0724007	

---

Рис. 8.2. Примеры регистрационных данных в файле BANK\_FILE.

Структуры разных видов, имеющие различные значения дискриминанта REC\_KIND, показаны на рис. 8.2. Продолжим текст пакета:

```

package BANK_IO is new DIRECT_IO ( BANK_REC ) ;
use BANK_IO ;
BANK_FILE : BANK_IO.FILE_TYPE ;
GLOBAL_NO_OF_BANKS : POSITIVE_COUNT ;
procedure RETRIEVE ( FORM_BANK_NAME : in out STRING ;
                    FORM_POS       : out COUNT ;
                    FORM_FOUND     : out BOOLEAN ) ;
end BANK_RESOURCES ;

```

Далее запишем текст тела пакета BANK\_RESOURCES:

```

package body BANK_RESOURCES is
  LOCAL_BANK_REC : BANK_REC;
  -- При конкретизации пакета ввода-вывода
  -- для файла с прямой или последовательной
  -- организацией можно использовать
  -- только лишь один тип данных. Однако
  -- если применить комбинированный тип с
  -- вариантной частью, то будет можно ис-

```

```
-- пользоваться несколько видов структур
-- данных. Для файла BANK_FILE следует
-- учитывать размеры структур с целью
-- экономии памяти. Как правило, следует
-- добиваться того, чтобы вариант структуры,
-- имеющий наибольший размер, использовался
-- в файле наиболее часто.
-- Структура LOCAL_BANK_REC с дискриминантом,
-- равным POSTING_DATA, будет, вероятно,
-- занимать максимальную часть объема файла
-- BANK_FILE, и поэтому количество компонент у
-- типа POST_INFO следует подбирать таким обра-
-- зом, чтобы компонента структуры POSTING_LINE
-- имела наибольший размер. Предопределенный
-- атрибут Ады SIZE (РАЗМЕР) дает минимальный
-- размер (в битах), требуемый для размещения
-- любого возможного объекта заданного типа.
-- Пример запроса этого атрибута:
-- BANK_REC'SIZE. Размер элемента файла будет
-- зависеть от конкретной версии языка Ада.
-- Он примерно равен значению, вырабатываемому
-- атрибутом SIZE.
```

```
procedure RETRIEVE(
    FORM_BANK_NAME : in out STRING;
    FORM_POS       : out COUNT;
    FORM_FOUND      : out BOOLEAN ) is
-- Эта процедура осуществляет поиск такого
-- элемента в файле BANK_FILE, название бан-
-- ка для которого совпадает с названием
-- банка в строке FORM_BANK_NAME. Если наз-
-- вание будет найдено, то переменная
-- FORM_FOUND получит значение TRUE, а в
-- противном случае эта переменная будет
-- иметь значение FALSE. Метод поиска,
-- применяемый в данной процедуре, назы-
-- вается двоичным поиском.
    LEFT_LIM : COUNT := 2;
    MIDDLE_NDX, RIGHT_LIM : COUNT;
begin
    RIGHT_LIM := GLOBAL_NO_OF_BANKS;
    FORM_FOUND := FALSE;
    loop
        MIDDLE_NDX := (LEFT_LIM + RIGHT_LIM)/2;
        READ(FILE => BANK_FILE,
            ITEM => LOCAL_BANK_REC,
            FROM => MIDDLE_NDX);
        if LOCAL_BANK_REC.MAST_HEAD.BANK_NAME =
            FORM_BANK_NAME
        then
            FORM_FOUND := TRUE;
            FORM_POS := MIDDLE_NDX;
            exit;
        elsif LOCAL_BANK_REC.MAST_HEAD.BANK_NAME <
            FORM_BANK_NAME
        then
            LEFT_LIM := MIDDLE_NDX;
        else
            RIGHT_LIM := MIDDLE_NDX;
        end if;
    end loop;
end RETRIEVE;
```

```

    end if;
    if LEFT_LIM >= RIGHT_LIM
    then
        FORM_POS := RIGHT_LIM;
        exit;
    end if;
end loop;
end RETRIEVE;
-- Далее располагается текст, с помощью
-- которого выполняется инициализация пере-
-- менных пакета.
begin
    OPEN(FILE => BANK_FILE,
        MODE => INOUT_FILE,
        -- Требуется и чтение, и запись.
        NAME => "BANK_MASTER.DAT",
        FORM => " " );
    -- Для некоторых компиляторов Ады при со-
    -- здании файла (если тип элементов
    -- файла - не уточненный как в данном случае)
    -- необходимо указывать в качестве части фак-
    -- тического параметра FORM максимальный размер
    -- структуры для элементов файла.
    READ(FILE => BANK_FILE,
        ITEM => LOCAL_BANK_REC,
        FROM => 1);
    -- Прочитать первый элемент файла BANK_FILE,
    -- чтобы определить количество банков.
    GLOBAL_NO_OF_BANKS := POSITIVE_COUNT(
        LOCAL_BANK_REC.NO_BANKS);
end BANK_RESOURCES;
```

Теперь представим текст самой программы.

### Программа BANK\_MAINT

```

with TEXT_IO; use TEXT_IO;
-- Если пакет BANK_RESOURCES оттранслирован,
-- то фразы, располагающиеся ниже, делают
-- его непосредственно видимым.
with BANK_RESOURCES; use BANK_RESOURCES;
--
procedure BANK_MAINT is
    CURR_IN_REC : IN_REC;
    ACT_POS : BANK_IO.COUNT;
    ACT_FOUND : BOOLEAN;
    CURR_BANK_REC, SAVE_BANK_REC : BANK_REC;
begin
    -- Для случаев, когда переменная
    -- CURR_IN_REC.IN_CODE равна 'A' или 'C',
    -- лучше было бы использовать некоторые
    -- дополнительные программы из пакета
    -- BANK_RESOURCES. Эта ситуация рассматри-
    -- вается в упр.3 в конце данной главы.
    if not BANK_IO.IS_OPEN (BANK_FILE);
    -- Заметьте, что и в пакете TEXT_IO, и
    -- в пакете BANK_IO есть функция
```

```

-- IS_OPEN и процедура OPEN.
then
  BANK_IO.OPEN(
    FILE => BANK_FILE,
    MODE => BANK_IO.INOUT_FILE,
    NAME => "BANK_MASTER.DAT",
    FORM => "" );
end if;
GET( CURR_IN_REC.IN_CODE );
while CURR_IN_REC.IN_CODE /= 'Z'
loop
  GET(CURR_IN_REC.IN_DATA.BANK_NAME);
  GET(CURR_IN_REC.IN_DATA.OTHER_INFO);
  case CURR_IN_REC.IN_CODE is
    when 'I' =>
      RETRIEVE(CURR_IN_REC.IN_DATA.BANK_NAME,
        ACT_POS, ACT_FOUND );
      if ACT_FOUND
        then
          PUT(" Bank found ");
        else
          PUT(" Bank not found ");
        end if;
      when 'A' =>
        -- Это - неэффективный способ обновления
        -- банковского файла. Файл, однако, об-
        -- новляется очень редко. Здесь этот спо-
        -- соб дает возможность поупражняться в
        -- использовании ряда подпрограмм из пакета
        -- DIRECT_IO.
        for I in 2 .. GLOBAL_NO_OF_BANKS
          loop
            READ(FILE => BANK_FILE,
              ITEM => CURR_BANK_REC,
              FROM => I );
          exit when
            CURR_IN_REC.IN_DATA.BANK_NAME <
            CURR_BANK_REC.MAST_HEAD.BANK_INFO.BANK_NAME;
          end loop;
          -- После считывания данных индекс
          -- увеличивается на единицу.
          ACT_POS := COUNT(INDEX(BANK_FILE))-1;
          SAVE_BANK_REC := (MASTER_DATA,
            CURR_IN_REC, 0, 0 1);
          -- Это - присваивание позиционного
          -- агрегата.
        for I in POSITIVE_COUNT(ACT_POS) ..
          GLOBAL_NO_OF_BANKS + 1
          -- Этот цикл перемещает структуры
          -- после вставки.
          loop
            WRITE( FILE => BANK_FILE,
              ITEM => SAVE_BANK_REC,
              TO => I );
            SAVE_BANK_REC := CURR_BANK_REC;
            READ( FILE => BANK_FILE,
              ITEM => CURR_BANK_REC );

```



```

end loop;
-- Обновить количество банков,
-- записанное в первом элементе
-- файла.
READ(BANK_FILE, CURR_BANK_REC, 1);
CURR_BANK_REC.NO_BANKS :=
    POSITIVE(GLOBAL_NO_OF_BANKS) + 1;
WRITE( BANK_FILE, CURR_BANK_REC, 1);
when 'C' =>
    RETRIEVE(CURR_IN_REC.IN_DATA.BANK_NAME,
              ACT_POS, ACT_FOUND);
    if ACT_FOUND
    then
        READ(FILE => BANK_FILE,
              ITEM => CURR_BANK_REC,
              FROM => POSITIVE_COUNT(ACT_POS));
        CURR_BANK_REC.MAST_HEAD :=
            CURR_IN_REC.IN_DATA;
        WRITE(FILE => BANK_FILE,
              ITEM => CURR_BANK_REC,
              TO   => ACT_POS );
    else
        PUT(" No such bank ");
    end if;
when 'Z' => exit;
when others => null;
end case;
SKIP_LINE;
GET(CURR_IN_REC.IN_CODE);
end loop;
BANK_IO.CLOSE(BANK_FILE);
end BANK_MAINT;

```

### 8.3.3. Преимущества файлов прямого доступа

В данном разделе на примере двух программ будут проиллюстрированы преимущества использования файлов прямого доступа вместо последовательных файлов при произвольном порядке обращения к элементам файла и обработке этих элементов. Здесь предполагается, что доступен файл `BANK_FILE` из предыдущего подраздела.

Первая программа регистрирует сделки маклера, работающего с иностранной валютой. Сведения о каждой сделке содержатся в отдельной входной строке, имеющей формат:

Позиции	Данные
1-20	BUYER (покупатель) — название банка из файла <code>BANK_FILE</code>
21-40	SELLER (продавец) — название банка из файла <code>BANK_FILE</code>
41-43	CURR_BOUGHT — символы, обозначающие покупаемую валюту, например US\$ (доллары США), DM (марки), SF (швейцарские франки)
44-46	CURR_SOLD — символы, обозначающие продаваемую валюту
47-55	X_RATE — курс обмена
56-61	VALUE_DATE — дата заключения сделки
62-67	TRADE_DATE — дата, когда деньги переходят от одного владельца к другому. Обычно это следующий рабочий день, расположенный после даты VALUE_DATE

Признаком конца входных данных служит строка, в которой поле `BUYER` имеет значение «12345678901234567890». Маклеру причитается комиссионное вознаграждение

в размере 25 долл. на каждый миллион долларов сделки. Сумма вознаграждения подсчитывается, но не регистрируется.

Каждая входная строка проходит проверку на корректность содержащихся в ней данных, т.е. проверяются даты и названия банков. Если строка содержит правильные сведения о сделке, то она помещается в соответствующий файл с данными о сделках. В противном случае выдается предупреждающее сообщение. Если данные о сделке корректны, то генерируется полный номер сделки. Он состоит из номера месяца, дня и номера сделки за этот день. Полный номер сделки используется для определения файла и индекса элемента этого файла, содержащего сведения о данной сделке. Перед тем как представить текст программы, приведем спецификацию пакета с необходимыми объявлениями.

```
with DIRECT_IO;
package TRANSACTION_RESOURCES is
  type TRANS_HEADER is
    record
      DAY_TRANS_NO : NATURAL;
      FIRST_TRANS  : NATURAL;
      LAST_TRANS   : NATURAL;
    end record;
  type MONTH_HEADER is array (1 .. 31)
    of TRANS_HEADER;
  type TRANS_FILE_HEADER is
    record
      YEAR_N_MONTH : STRING(1 .. 4);
      TRANS_STATUS : MONTH_HEADER;
    end record;
  type SHORT_DATE is
    record
      SHORT_YY : INTEGER range 0 .. 99;
      SHORT_MM : INTEGER range 1 .. 12;
      SHORT_DD : INTEGER range 1 .. 31;
    end record;
  type IN_TRANS is
    record
      BUYER   : STRING(1 .. 20);
      SELLER  : STRING(1 .. 20);
      CURR_BOUGHT : STRING(1 .. 3);
      CURR_SOLD  : STRING(1 .. 3);
      X_RATE    : FLOAT;
      VALUE_DATE : SHORT_DATE;
      TRADE_DATE : SHORT_DATE;
    end record;
  type TRANS_INFO is
    record
      TRANS_NO : NATURAL;
      TRANS_BODY : IN_TRANS;
      TRANS_NEXT : NATURAL;
    end record;
  type TRANS_ELEM_KIND is (TOP_LINE, REC_LINE);
  type TRANS_REC (REC_KIND : TRANS_ELEM_KIND :=
    REC_LINE) is
    record
      case REC_KIND is
        when TOP_LINE =>
          HEADER_LINE : TRANS_FILE_HEADER;
          -- Может оказаться, что размер строки
```

```

-- заголовок окажется слишком большим.
-- В упражнениях в конце данной главы
-- рассматривается другое построение
-- структур.
when REC_LINE =>
  TR_LINE : TRANS_INFO;
end case;
end record;

```

Пример файла TRANS\_FILE и некоторых его элементов дан на рис. 8.3. Продолжим текст пакета:

```

package TRANS_IO is new DIRECT_IO(TRANS_REC);
use TRANS_IO;
TRANS_FILE : TRANS_IO.FILE_TYPE;
-- Предполагается, что создаются файлы со сведе-
-- ниями о сделках (возможно, в программе из
-- разд. 7.1), даже если размер этих файлов равен
-- нулю.
end TRANSACTION_RESOURCES;

```

В следующей программе употребляется версия процедуры PUT, входящая в конкретизированный пакет INT\_IO. Объявление этой процедуры:

```

procedure PUT (TO : out STRING; ITEM : NUM;
               BASE : in NUMBER_BASE :=
               DEFAULT_BASE);

```

Эта процедура помещает значение фактического параметра, соответствующего формальному параметру ITEM, в строковую переменную, соответствующую формальному параметру TO (а не в файл с режимом обмена информацией OUT\_FILE).

Процедура GET из пакета INT\_IO имеет объявление:

```

procedure GET (FROM : in STRING;
               ITEM : out NUM;
               LAST : out NATURAL);

```

В противоположность соответствующей процедуре PUT данная процедура GET выполняет чтение целого значения из строки, соответствующей формальному параметру FROM. Значение, которое получит переменная, соответствующая формальному

---

Первый элемент файла TRANS\_FILE принадлежит к типу

```

TRANS_REC (TOP_LINE);
TOP_LINE

```

YEAR\_N\_MONTH

TRANS\_STATUS

	(1)	(2)	.. (23)	.. (31)
8607	01 002 045	02 008 075	23 021 089	000

Все остальные элементы файла относятся к типу

TRANS\_REC (REC\_LINE). Например, 2- и 45-й элементы могут выглядеть так:

TRANS_NO	TRANS_BODY	TRANS_NEXT
001	CHEMICAL BANK ...	003
034	CHASE ...	000

---

Рис. 8.3. Примеры регистрационных записей в файле TRANS\_FILE.

параметру LAST,—это значение индекса<sup>1)</sup> последнего прочитанного символа. Данная разновидность процедуры GET будет использована во второй программе настоящего раздела.

Предположим теперь, что пакет TRANSACTION\_RESOURCES уже оттранслирован. Тогда текст первой программы будет таким:

#### Программа CURR\_TRANSACTION\_PROC

```
with TEXT_IO; use TEXT_IO;
with CHECK_DATES_ALT; use CHECK_DATES_ALT;
with BANK_RESOURCES; use BANK_RESOURCES;
with TRANSACTION_RESOURCES;
use TRANSACTION_RESOURCES;
with LEGAL_HOLIDAYS; use LEGAL_HOLIDAYS;
-- Пакет LEGAL_HOLIDAYS был определен в гл.7.
-- В нем используется пакет CHECK_DATES_ALT.
-- Предполагается, что эти пакеты уже оттран-
-- слированы.
--
procedure CURR_TRANSACTION_PROC is
  CURR_IN_TRANS : IN_TRANS;
  CURR_TRANS_REC, SAVE_TRANS_REC : TRANS_REC;
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  package FLT_IO is new FLOAT_IO(FLOAT);
  use FLT_IO;
  BANK_POS : POSITIVE_COUNT;
  BANK_FOUND, DATE_VALID : BOOLEAN;
  CURR_DATE : DATE;
  --
  procedure MOVE_SHRT_DATE_TO_REG_DATE
    (FORM_SH_DATE : in SHORT_DATE;
     FORM_LG_DATE : out DATE) is
  begin
    FORM_LG_DATE.YEAR_NO := FORM_SH_DATE.SHORT_YY;
    FORM_LG_DATE.MONTH_NO := FORM_SH_DATE.SHORT_MM;
    FORM_LG_DATE.DAY_NO := FORM_SH_DATE.SHORT_DD;
  end MOVE_SHRT_DATE_TO_REG_DATE;
  --
  procedure INSERT_REC(FORM_IN_TRANS : IN_TRANS) is
    EXT_NAME : STRING(1 .. 10);
    YY_STR, MM_STR : STRING(1 .. 2);
    LOC_1, LOC_OTHER : TRANS_REC;
    LOCAL_MONTH_HEADER : MONTH_HEADER :=
      (1 .. 31 => (0, 1, 1));
    WRK_LAST_TR : NATURAL;
    WRK_CURR_TR : NATURAL;
  begin
    -- Вначале определим имя внешнего файла и откроем
    -- этот файл.
    PUT(YY_STR, FORM_IN_TRANS.VALUE_DATE.SHORT_YY);
    -- Этот оператор преобразовывает целые значения
    -- в строковые.
    -- Далее заменим пробел, стоящий перед числом на
    -- '0'.
    if YY_STR(1) = ' '
```

<sup>1)</sup> То есть позиции в строке.—Прим. перев.

```

    then
        YY_STR(1) := '0';
    end if;
    PUT(MM_STR, FORM_IN_TRANS.VALUE_DATE.SHORT_MM);
    if M_STR(1) = ' '
    then
        MM_STR(1) := '0';
    end if;
    EXT_NAME := "FY" & YY_STR & MM_STR & ".DAT";
    if TRANS_IO.IS_OPEN (TRANS_FILE)
    then
        TRANS_IO.CLOSE(TRANS_FILE);
    end if;
    TRANS_IO.OPEN ( FILE => TRANS_FILE,
                    MODE => INOUT_FILE,
                    NAME => EXT_NAME,
                    FORM => "");
    if TRANS_IO.SIZE (TRANS_FILE) < 1
    -- Это условие будет истинно, если в файл
    -- ничего не записано.
    then
        -- Запишем первый элемент файла.
        LOC_1 := (TOP_LINE,
                 (YY_STR & MM_STR, LOCAL_MONTH_HEADER));
        TRANS_IO.WRITE(TRANS_FILE, LOC_1, 1);
    end if;
    TRANS_IO.READ(TRANS_FILE, LOC_1, 1);
    -- В тексте программы, расположенном ниже,
    -- целесообразно использовать переименование
    -- процедур (см. упр.6 в конце главы).
    WRK_LAST_TR :=
        LOC_1.HEADER_LINE.TRANS_STATUS.LAST_TRANS
        (FORM_IN_TRANS.VALUE_DATE.SHORT_DD);
    if WRK_LAST_TR /= 1
    then
        -- Для данного дня месяца есть еще сделки,
        -- последняя из них ставится в конец.
        TRANS_IO.READ ( TRANS_FILE, LOC_OTHER,
                        WRK_LAST_TR);
        -- Обновить следующую запись.
        LOC_OTHER.TR_LINE.TRANS_NEXT :=
            POSITIVE(TRANS_IO.SIZE(TRANS_FILE))+1;
        TRANS_IO.WRITE ( TRANS_FILE, LOC_OTHER,
                        WRK_LAST_TR);
    else
        -- Инициализировать первую сделку дня.
        LOC_1.HEADER_LINE.TRANS_STATUS.FIRST_TRANS
        (FORM_IN_TRANS.VALUE_DATE.SHORT_DD) :=
            POSITIVE(TRANS_IO.SIZE(TRANS_FILE))+1;
    end if;
    WRK_CURR_TR :=
        LOC_1.HEADER_LINE.TRANS_STATUS.DAY_TRANS_NO
        (FORM_IN_TRANS.VALUE_DATE.SHORT_DD) + 1;
    LOC_1.HEADER_LINE.TRANS_STATUS.DAY_TRANS_NO
        (FORM_IN_TRANS.VALUE_DATE.SHORT_DD) :=
            WRK_CURR_TR;
    WRK_LAST_TR :=

```

```

    POSITIVE(TRANS_IO.SIZE(TRANS_FILE))+1;
    LOC_1.HEADER_LINE.TRANS_STATUS.LAST_TRANS
    (FORM_IN_TRANS.VALUE_DATE.SHORT_DD) :=
    WRK_LAST_TR;
    -- Затем обновляется первый элемент файла.
    TRANS_IO.WRITE(TRANS_FILE, LOC_1, 1);
    LOC_OTHER :=
    (REC_LINE, (WRK_CURR_TR, FORM_IN_TRANS, 0));
    -- Далее запишем данные о новой сделке.
    TRANS_IO.WRITE(TRANS_FILE, LOC_OTHER, WRK_LAST_TR);
    TRANS_IO.CLOSE(TRANS_FILE);
end INSERT_REC;

--
begin
    -- Заметьте, что при обработке пакета BANK_RESOURCES
    -- файл BANK_FILE будет открыт (обработка пакета
    -- включает выполнение его тела).
    GET(CURR_IN_TRANS.BUYER);
    while CURR_IN_TRANS.BUYER /= "12345678901234567890"
    loop
        RETRIEVE(CURR_IN_TRANS.BUYER, BANK_POS, BANK_FOUND);
        if BANK_FOUND
        then
            GET(CURR_IN_TRANS.SELLER);
            RETRIEVE(CURR_IN_TRANS.SELLER, BANK_POS,
                BANK_FOUND);
        end if;
        if BANK_FOUND
        then
            -- Если переменная BANK_FOUND имеет значение
            -- TRUE, то покупатель и продавец имеют пра-
            -- вильные обозначения.
            GET(CURR_IN_TRANS.CURR_BOUGHT);
            GET(CURR_IN_TRANS.CURR_SOLD);
            GET(CURR_IN_TRANS.X_RATE);
            GET(CURR_IN_TRANS.VALUE_DATE.SHORT_YY);
            GET(CURR_IN_TRANS.VALUE_DATE.SHORT_MM);
            GET(CURR_IN_TRANS.VALUE_DATE.SHORT_DD);
            GET(CURR_IN_TRANS.TRADE_DATE.SHORT_YY);
            GET(CURR_IN_TRANS.TRADE_DATE.SHORT_MM);
            GET(CURR_IN_TRANS.TRADE_DATE.SHORT_DD);
            MOVE_SHRT_DATE_TO_REG_DATE(
                CURR_IN_TRANS.VALUE_DATE, CURR_DATE);
            FILL_IN_DATE(CURR_DATE, DATE_VALID);
            if DATE_VALID
            then
                DATE_VALID := not IS_LEGAL_HOLIDAYS
                    (CURR_DATE);
            end if;
            if DATE_VALID
            then
                MOVE_SHRT_DATE_TO_REG_DATE
                    (CURR_IN_TRANS.TRADE_DATE, CURR_DATE);
                FILL_IN_DATE(CURR_DATE, DATE_VALID);
                if DATE_VALID
                then
                    DATE_VALID := not

```

```

        IS_LEGAL_HOLIDAYS(CURR_DATE);
    end if;
end if;
end if;
SKIP_LINE;
if DATE_VALID and BANK_FOUND
then
    -- В строке представлены верные данные о
    -- сделке, и эта строка будет записана в
    -- файл с данными о сделках.
    INSERT_REC(CURR_IN_TRANS);
else
    PUT(" Invalid transaction ");
end if;
GET(CURR_IN_TRANS.BUYER);
end loop;
BANK_IO.CLOSE(BANK_FILE);
end CURR_TRANSACTION_PROC;

```

Вторая программа, текст которой представлен ниже, осуществляет в файле BANK\_FILE регистрацию сделок, сведения о которых содержатся в разных файлах, несущих информацию о сделках. Таким образом, будут зарегистрированы сделки для каждого банка. Предполагается, что входные данные состоят из одной строки, содержащей номер года (он должен быть равен 1986) и месяца (две цифры). Считается, что доступны файл BANK\_FILE и файлы, несущие информацию о сделках (начиная с 1986 г.).

### Программа POSTING\_PROC

```

with TEXT_IO; use TEXT_IO;
with TRANSACTION_RESOURCES;
use TRANSACTION_RESOURCES;
with BANK_RESOURCES; use BANK_RESOURCES;
-- Предполагается, что эти пакеты уже от-
-- транслированы.
procedure POSTING_PROC is
    package INT_IO is new INTEGER_IO(INTEGER);
    use INT_IO;
    CURR_TRANS_REC_1, CURR_TRANS_REC_OTHER :
        TRANS_REC;
    YY_AND_MM : STRING(1 .. 4);
    MM_ONLY : STRING(1 .. 2);
    EXT_NAME : STRING(1 .. 8);
    WORK_POSITIVE : NATURAL;
    CURR_TRANS_HEADER : TRANS_HEADER;
    CURR_TRANS_INDEX : POSITIVE_INDEX;
    procedure POST_INDIV_BANK(ANY_TRANS : TRANS_NUMBER;
        ANY_BANK : STRING) is
        BK_POS, TR_POS, SAVE_TR_POS : COUNT;
        BK_FOUND : BOOLEAN;
        ANY_BANK_REC : BANK_REC;
        ANY_POST_REC : BANK_REC;
        WRK_POST_INFO : POST_INFO;
    begin
        RETRIEVE(ANY_BANK, BK_POS, BK_FOUND);
        if BK_FOUND
            then

```

```

BANK_IO.READ(BANK_FILE, ANY_BANK_REC, BK_POS);
if ANY_BANK_REC.FIRST_TRAN = 0
-- Это условие будет истинным, если для дан-
-- ного конкретного банка сделки пока еще
-- не регистрировались.
then
if BANK_IO.SIZE(BANK_FILE) <=
GLOBAL_NO_OF_BANKS
-- Данное условие будет истинным, если ни
-- для каких банков сделки не регистри-
-- ровались.
then
TR_POS := GLOBAL_NO_OF_BANKS + 1;
else
TR_POS := BANK_IO.SIZE(BANK_FILE) + 1;
end if;
ANY_BANK_REC.FIRST_TRAN := POSITIVE(TR_POS);
ANY_BANK_REC.LAST_TRAN := POSITIVE(TR_POS);
ANY_BANK_REC.LAST_POS := 1;
WRK_POST_INFO := (ANY_TRANS,
2 .. 12 => (1,1,1));
ANY_POST_REC :=
(POSTING_DATA, WRK_POST_INFO, 0);
-- Инициализировать регистрационную строку с
-- информацией о сделке.
else
if ANY_BANK_REC.LAST_POS = 12
-- Если это условие истинно, то строка с ре-
-- гистрационными данными заполнена до конца.
then
TR_POS := SIZE(BANK_FILE) + 1;
BANK_IO.READ(BANK_FILE, ANY_POST_REC,
ANY_BANK_REC.LAST_TRAN);
ANY_POST_REC.NXT_PST_LINE :=
POSITIVE(TR_POS);
BANK_IO.WRITE(BANK_FILE, ANY_POST_REC,
ANY_BANK_REC.LAST_TRAN);
ANY_BANK_REC.LAST_TRAN :=
POSITIVE(TR_POS);
ANY_BANK_REC.LAST_POS := 1;
WRK_POST_INFO := (ANY_TRANS,
2 .. 12 => (1,1,1));
ANY_POST_REC :=
(POSTING_DATA, WRK_POST_INFO, 0);
else
-- Место есть.
TR_POS := ANY_BANK_REC.LAST_TRAN;
BANK_IO.READ(BANK_FILE, ANY_POST_REC,
ANY_BANK_REC.LAST_TRAN);
ANY_BANK_REC.LAST_POS :=
ANY_BANK_REC.LAST_POS + 1;
ANY_POST_REC.POSTING_LINE
( ANY_BANK_REC.LAST_POS ) :=
ANY_TRANS;
end if;
end if;
BANK_IO.WRITE(BANK_FILE, ANY_BANK_REC, BK_POS);
BANK_IO.WRITE(BANK_FILE, ANY_POST_REC, TR_POS);

```



```

else
  PUT(" Bank not found ");
  PUT ( ANY_BANK );
end if;
-- Заметьте, что сделка может быть зарегистри-
-- рована в ведомости только лишь одного бан-
-- ка, если обозначение партнера данного банка
-- не найдено.
and POST_INDIV_BANK;
--
procedure POST_BNK(FORM_TRANS_REC : TRANS_REC;
                   FORM_DAY : NATURAL) is
  LOCAL_FULL_TRANS_NO : TRANS_NUMBER;
  -- Данная процедура осуществляет подготовку
  -- к записи регистрационной информации для
  -- обоих банков.
begin
  LOCAL_FULL_TRANS_NO.DAY_PART := FORM_DAY;
  LOCAL_FULL_TRANS_NO.NO_PART :=
    FORM_TRANS_REC.TR_LINE.TRANS_NO;
  GET(FROM => MM_ONLY, ITEM =>
    LOCAL_FULL_TRANS_NO.NO_PART,
    LAST => WORK_POSITIVE );
  POST_INDIV_BANK (LOCAL_FULL_TRANS_NO,
    FORM_TRANS_REC.TR_LINE.TRANS_BODY.BUYER);
  POST_INDIV_BANK (LOCAL_FULL_TRANS_NO,
    FORM_TRANS_REC.TR_LINE.TRANS_BODY.SELLER);
end POST_BNK;
--
begin
  GET (YY_AND_MM);
  MM_ONLY := YY_AND_MM ( 3 .. 4 );
  if YY_AND_MM < "8601" or YY_AND_MM > "8612"
  then
    PUT (" Bad date ");
  else
    EXT_NAME := "FY" & YY_AND_MM & ".DAT";
    if not TRANS_IO.IS_OPEN(TRANS_FILE)
    then
      TRANS_IO.OPEN(FILE => TRANS_FILE,
        MODE => INOUT_FILE,
        NAME => EXT_NAME,
        FORM => "");
      TRANS_IO.READ(TRANS_FILE, CURR_TRANS_REC_1, 1);
    end if;
    for I in 1 .. 31
      loop
        CURR_TRANS_HEADER :=
          CURR_TRANS_REC_1.HEADER_LINE.TRANS_STATUS(I);
        CURR_TRANS_INDEX :=
          CURR_TRANS_HEADER.FIRST_TRANS;
        if CURR_TRANS_INDEX /= 1
        then
          TRANS_IO.READ(TRANS_FILE, CURR_TRANS_REC_OTHER,
            CURR_TRANS_INDEX);
          while CURR_TRANS_REC_OTHER.TR_LINE.TRANS_NEXT
            /= 0

```

```

loop
  POST_BNK(CURR_TRANS_REC_OTHER, 1);
  CURR_TRANS_INDEX :=
    CURR_TRANS_REC_OTHER.TR_LINE.TRANS_NEXT;
  TRANS_IO.READ(TRANS_FILE,
    CURR_TRANS_REC_OTHER, CURR_TRANS_INDEX);
end loop;
POST_BNK(CURR_TRANS_REC_OTHER, 1);
end if;
end loop;
end if;
if TRANS_IO.IS_OPEN(TRANS_FILE)
then
  TRANS_IO.CLOSE(TRANS_FILE);
end if;
if TRANS_IO.IS_OPEN(BANK_FILE)
then
  TRANS_IO.CLOSE(BANK_FILE);
end if;
end POSTING_PROC;

```

Следует ожидать, что в конкретных реализациях Ады будут иметься и другие пакеты, предназначенные для разнообразной обработки файлов. Например, можно будет встретить пакеты, осуществляющие работу с индексно-последовательными файлами, или пакеты, позволяющие выполнять ввод-вывод значений для «смеси» типов<sup>1)</sup>.

## 8.4. ОБРАБОТКА ФАЙЛОВ С ПОМОЩЬЮ ПАКЕТА TEXT\_IO

Пакет TEXT\_IO содержит средства, предназначенные для выполнения ввода-вывода в удобном для человека виде. В предыдущих главах уже были описаны и использовались многие из средств пакета TEXT\_IO. В данном разделе будут описаны некоторые из оставшихся подпрограмм пакета TEXT\_IO, в особенности средства выдачи листингов отчетов.

После того как TEXT\_IO файл будет открыт (или создан), становится возможным с помощью процедуры GET считывать данные из входных файлов. Запись данных в выходные файлы выполняется при помощи процедуры PUT.

В состав пакета TEXT\_IO, как было показано в разд. 7.5, входит несколько других родовых пакетов. Если потребуется считывать и записывать значения целого, перечисляемого или плавающего типов, то эти родовые пакеты следует соответствующим образом конкретизировать. Каждый из данных родовых пакетов содержит свои версии процедур GET и PUT, примеры употребления которых были даны во многих программах из предыдущих глав.

Некоторые подпрограммы из пакета TEXT\_IO имеют несколько разновидностей. В частности, для некоторых из подпрограмм могут существовать варианты с наличием формального параметра FILE и с отсутствием этого параметра. Если параметр FILE не задается при вызове подпрограммы, то используется принимаемый по умолчанию входной или выходной файл. Для каждой выполняющейся программы на Аде есть принимаемый по умолчанию входной и выходной файл. После начала выполнения программы такими подразумеваемыми файлами становятся принятые в данной операционной системе стандартный входной и стандартный выходной файлы. Стандартный

<sup>1)</sup> То есть файл будет содержать значения разных типов, а не одного конкретного типа. — *Прим. перев.*

входной файл открывается в режиме обмена информацией `IN_FILE`, а стандартный выходной файл – в режиме `OUT_FILE`. Эти положения имели силу для каждой из предыдущих программ.

Пакет `TEXT_IO` также содержит процедуры `SET_INPUT` и `SET_OUTPUT`. Они дают возможность установить в качестве файлов, выбираемых в данный момент времени по умолчанию, любые другие (открытые) текстовые файлы. Объявления этих процедур:

```
procedure SET_INPUT (FILE : in FILE_TYPE);
procedure SET_OUTPUT (FILE : in FILE_TYPE);
```

Имена файлов – стандартного входного, стандартного выходного, текущего входного и текущего выходного – можно определить путем вызова следующих функций:

```
function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;
function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;
```

Стандартный входной и стандартный выходной файлы могут относиться к одному и тому же физическому устройству, например, к терминалу. Имена этих файлов системно-зависимы. Типичными именами могут быть `SYSS$OUTPUT` или `SYSS$INPUT`.

Выходные `TEXT_IO`-файлы рассматриваются как последовательности символов, образующих строки. Позиции символов в строке определяются как номера колонок; строки объединяются в страницы. Конец строки помечается специальным признаком конца строки, а конец страницы – признаками конца строки и конца страницы. В конце файла располагаются признаки конца строки, конца страницы и конца файла. Пользователь не может воздействовать на эти признаки, фактическое их построение зависит от конкретной реализации языка Ада.

Размеры строк и страниц устанавливаются с помощью процедур `SET_LINE_LENGTH` и `SET_PAGE_LENGTH`. Объявления этих процедур:

```
procedure SET_LINE_LENGTH (FILE : in FILE_TYPE;
                           TO : in COUNT);
procedure SET_LINE_LENGTH (TO : in COUNT);
```

Эта версия относится к выходному файлу, принимаемому по умолчанию.

```
procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE;
                           TO : in COUNT);
procedure SET_PAGE_LENGTH (TO : in COUNT);
```

Здесь `COUNT` – целый тип, охватывающий значения от нуля до определяемого реализации целого числа.

Функции `LINE_LENGTH` и `PAGE_LENGTH` дают возможность узнать максимальный размер строки или страницы заданного выходного текстового файла. Объявления этих функций даны в приложении В.

В пакете `TEXT_IO` есть и ряд других подпрограмм, которые можно применить при работе с номерами колонок, строк и страниц. Помимо процедур `NEW_LINE` и `SKIP_LINE`, которые использовались в предыдущих программах, можно воспользоваться и рядом других подпрограмм. Приведем их объявления:

```
function END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;
```

Эта функция даст значение `TRUE`, если в следующей позиции во входном файле будет обнаружен признак конца строки или конца файла.

```
function END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
```

Эта функция вырабатывает значение TRUE, если в следующей позиции обнаруживается признак конца строки и конца страницы или же признак конца файла.

```
function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;  
function END_OF_FILE return BOOLEAN;
```

Данная функция даст значение TRUE, если, начиная со следующей позиции входного файла, будет располагаться последовательность признаков конца строки, конца страницы и конца файла или же только признак конца файла.

Текущие номера колонок, строк и страниц можно узнать с помощью следующих функций:

```
function COL (FILE : in FILE_TYPE) return POSITIVE_COUNT;  
function COL return POSITIVE_COUNT;
```

Тип POSITIVE\_COUNT является подтипом типа COUNT с положительными значениями.

```
function LINE (FILE : in FILE_TYPE) return POSITIVE_COUNT;  
function LINE return POSITIVE_COUNT;
```

Эта функция дает значение текущего номера строки в текущей странице. Система отслеживает и номер текущей страницы, значение которого можно получить при вызове функций:

```
function PAGE (FILE : in FILE_TYPE) return POSITIVE_COUNT;  
function PAGE return POSITIVE_COUNT;
```

С помощью следующих процедур можно управлять расположением колонок, строк и страниц:

```
procedure NEW_PAGE (FILE : in FILE_TYPE);  
procedure NEW_PAGE;
```

Эти процедуры записывают признак конца строки (если текущая строка еще не закончилась) и признак конца страницы. Предполагается, что файл – выходной. В процедурах

```
procedure SKIP_PAGE (FILE : in FILE_TYPE);  
procedure SKIP_PAGE;
```

предполагается, что файл – входной. Они считывают и пропускают все символы вплоть до признака конца страницы. Затем текущий номер страницы увеличивается на 1, а текущие номера колонки и строки устанавливаются равными 1. Процедуры:

```
procedure SET_COL (FILE : in FILE_TYPE;  
                  TO : in POSITIVE_COUNT);  
procedure SET_COL (TO : in POSITIVE_COUNT);
```

можно использовать как со входными, так и с выходными файлами. В качестве фактического параметра, согласующегося с формальным параметром TO, следует указывать новый номер колонки. Процедуры

```
procedure SET_LINE (FILE : in FILE_TYPE;  
                   TO : in POSITIVE_COUNT);  
procedure SET_LINE (TO : in POSITIVE_COUNT);
```

можно употреблять с файлами, режимы обмена информацией с которыми – IN\_FILE или OUT\_FILE. Новым номером строки будет значение, указываемое в качестве фактического параметра, соответствующего формальному параметру TO.

При некорректном применении данных подпрограмм могут возникать исключительные ситуации, которые поясняются в гл. 11.

## Monthly Statement for Bank : XXXXXXXXXXXXXXXXXXXX

		Month of YYMM			V_DATE	T_DATE
BUYER	SELLER	BOT	SOLD	CROSS_RATE		
XXXXXXXX	YYYYYYYY	US\$	DM	0.30123	MMDD	MMDD
ZZZZZZZZ	YYYYYYYY	US\$	SF	0.35422	MMDD	MMDD

Рис. 8.4. Пример формата отчета.

В следующей программе употребляются некоторые из введенных здесь подпрограмм, входящих в пакет TEXT\_IO. Программа составляет месячные отчеты для нескольких заданных банков путем выделения необходимой информации из файлов BANK\_FILE и TRANS\_FILE (см. предыдущий раздел). Формат отчета представлен на рис. 8.4.

## Программа REPORT\_GEN

```

with TEXT_IO; use TEXT_IO;
with TRANSACTION_RESOURCES;
use TRANSACTION_RESOURCES;
with BANK_RESOURCES; use BANK_RESOURCES;
-- Предполагается, что эти пакеты уже от-
-- транслированы.
procedure REPORT_GEN is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  package FLT_IO is new FLOAT_IO(FLOAT);
  use FLT_IO;
  DESIRED_BANK : STRING(1 .. 20);
  REPORT_FILE : TEXT_IO.FILE_TYPE;
  YY_AND_MM : STRING(1 .. 4);
  MM_ONLY : STRING(1 .. 2);
  EXT_NAME : STRING(1 .. 8);
  WORK_POSITIVE : NATURAL;
  BK_POS, TR_POS, SAVE_TR_POS : POSITIVE_COUNT;
  BK_FOUND, TRANS_FILE_FOUND : BOOLEAN;
  ANY_BANK_REC : BANK_REC;
  ANY_POST_REC : BANK_REC;
  procedure WRITE_HEADING is
  begin
    -- Запись в файл REPORT_FILE начинается с
    -- третьей строки.
    SET_LINE (TO => 3);
    SET_COL (TO => 55);
    PUT (PAGE);
    SET_LINE (TO => 5);
    SET_COL (TO => 5);
    PUT ("Monthly Statements for Bank ");
    PUT (DESIRED_BANK);
    SET_LINE (TO => 7);
    SET_COL (TO => 20);
    PUT (" Month of ");
    PUT (YY_AND_MM);
    SET_LINE (TO => 10);
    PUT (" BUYER ");
    SET_COL (TO => 21);
    PUT (" SELLER ");
  end WRITE_HEADING;
end REPORT_GEN;

```

```

SET_COL (TO => 41 "");
PUT (" BOT ");
PUT (" SOLD ");
PUT (" CROSS_RATE ");
PUT (" V-DATE ");
PUT (" T-DATE ");
SET_LINE (TO => 13);
end WRITE_HEADING;
procedure PRINT_TRAN (FORM_INFO : TRANS_INFO);
-- Эта процедура печатает строку с информацией
-- о сделке.
begin
  if LINE > 58
  then
    WRITE_HEADING;
  end if;
  PUT(FORM_INFO.BUYER);
  PUT(FORM_INFO.SELLER);
  SET_COL (TO => 43);
  PUT(FORM_INFO.CURR_BOUGHT);
  SET_COL (TO => 49);
  PUT(FORM_INFO.CURR_SOLD);
  PUT(FORM_INFO.X_RATE, 11, 2);
  SET_COL(62);
  PUT(FORM_INFO.VALUE_DATE.SHORT_DD, 2);
  SET_COL(65);
  PUT(FORM_INFO.TRADE_DATE.SHORT_DD, 2);
end PRINT_TRAN;
--
procedure GET_N_WRITE_FULL_TRAN
  (FORM_TRANS : TRANS_NUMBER) is
-- Данная процедура будет последовательно про-
-- сматривать сделки заданного дня и отобра-
-- зит информацию о них в файле REPORT_FILE.
-- В упр.9 в конце этой главы предлагается
-- воспользоваться более эффективным способом
-- поиска необходимой информации.
--
  LOCAL_TRANS_REC_1, LOCAL_TRANS_REC_OTHER :
    TRANS_REC;
  WRK_LAST_TR : POSITIVE_COUNT;
  WRK_CURR_TR : NATURAL;
begin
  TRANS_IO.READ(TRANS_FILE, LOCAL_TRANS_REC_1,
    1 );
  WRK_LAST_TR :=
    LOCAL_TRANS_REC_1.HEADER_LINE.TRANS_STATUS.
    LAST_TRANS ( FORM_TRANS.DAY_PART );
  WRK_CURR_TR :=
    LOCAL_TRANS_REC_1.HEADER_LINE.TRANS_STATUS.
    FIRST_TRANS ( FORM_TRANS.DAY_PART );
  TRANS_IO.READ (TRANS_FILE, LOCAL_TRANS_REC_OTHER,
    WRK_CURR_TR);
  loop
    if LOCAL_TRANS_REC_OTHER.REG_LINE.TRAN_NO =
      FORM_TRANS.NO_PART
    then
      PRINT_TRAN(LOCAL_TRANS_REC_OTHER.REG_LINE);

```

```

        end if;
        exit when WRK_CURR_TR = WRK_LAST_TR;
        WRK_CURR_TR := LOCAL_TRANS_REC_OTHER.REG_LINE.
            TRAN_NEXT;
        end loop;
    end GET_N_WRITE_FULL_TRAN;
-- Начало главной программы.
begin
-- Первоначально текущим файлом ввода служит стан-
-- дартный файл ввода.
-- Читать название требуемого банка.
    GET (DESIRED_BANK);
    RETRIEVE(DESIRED_BANK, BK_POS, BK_FOUND);
    if not BK_FOUND
    then
        PUT(" Bank not in Bank File ");
    end if;
-- Прочитать номер месяца в формате YYMM.
    GET (YY_AND_MM);
    MM_ONLY := YY_AND_MM ( 3 .. 4 );
    if YY_AND_MM < "8601" or YY_AND_MM > "8612"
    then
        PUT (" Bad date ");
        TRANS_FILE_FOUND := FALSE;
    else
        EXT_NAME := "FY" & YY_AND_MM & ".DAT";
        if not IS_OPEN(TRANS_FILE)
        then
            TRANS_IO.OPEN(FILE => TRANS_FILE,
                MODE => INOUT_FILE,
                NAME => EXT_NAME,
                FORM => "");
            TRANS_IO.READ(TRANS_FILE, CURR_TRANS_REC_1, 1);
            TRANS_FILE_FOUND := TRUE;
        end if;
    end if;
    if TRANS_FILE_FOUND and BK_FOUND
    then
        -- Создать файл, в который будет записываться
        -- отчет.
        TEXT_TO.CREATE(FILE => REPORT_FILE,
            MODE => OUT_FILE,
            NAME => "BANK_REPORT.DAT",
            FORM => "");
        -- Заменить файл, принимаемый по умолчанию, со
        -- стандартного файла на файл REPORT_FILE.
        SET_OUTPUT ( FILE => REPORT_FILE );
        -- Установить требуемые размеры строки и
        -- страницы.
        SET_LINE_LENGTH ( TO => 72 );
        -- Предыдущий оператор эквивалентен оператору
        -- SET_LINE_LENGTH (FILE=>REPORT_FILE,TO=>72);
        -- поскольку теперь файл, выбираемый по
        -- умолчанию, - это файл REPORT_FILE.
        SET_PAGE_LENGTH ( TO => 65 );
        -- Если сделки были уже зарегистрированы при
        -- помощи процедуры POSTING_PROC, то порядок

```

```

-- расположения в файле для таких сделок
-- будет соответствовать их порядку в
-- файле TRANS_FILE. В упр. 8 в конце данной
-- главы предлагается более общий подход к
-- регистрации сделок в файле BANK_FILE.
BANK_IO.READ(BANK_FILE, ANY_BANK_REC, BK_POS);
if ANY_BANK_REC.FIRST_TRAN = 0
  -- Это условие будет истинным, если для
  -- данного конкретного банка сделки пока
  -- еще не регистрировались.
  then
    PUT(" No transaction for this Bank ");
  else
    -- Начать с элемента, содержащего сведения
    -- о сделках, зарегистрированных в самом
    -- начале.
    CURR_POS := 1;
    TR_POS := ANY_BANK_REC.FIRST_TRAN;
    BANK_IO.READ ( BANK_FILE, ANY_POST_REC,
      *      TR_POS );
    loop
      -- Правильный ли номер у сделки ?
      if ANY_POST_REC.POSTING_LINE
        (CURR_POS ).MO_PART =
          CHARACTER'VAL(MM_ONLY) -
          CHARACTER'VAL ('0')
        then
          GET_N_WRITE_FULL_TRAN
            (ANY_POST_REC.POSTING_LINE(CURR_POS));
        end if;
        -- Следующее выражение будет истинным
        -- только после окончания обработки по-
        -- следней сделки данного банка.
        exit when CURR_POS = ANY_BANK_REC.LAST_POS
          and TR_POS = ANY_BANK_REC.LAST_TRAN;
        CURR_POS := CURR_POS + 1;
        if CURR_POS > 12
          then
            -- Должна быть считана следующая регист-
            -- рационная запись
            CURR_POS := 1;
            TR_POS := ANY_POST_REC.NXT_PST_LINE;
            BANK_IO.READ ( BANK_FILE,
              ANY_POST_REC, TR_POS );
          end if;
        end loop;
      end if;
      TEXT_IO.CLOSE (REPORT_FILE);
    end if;
    if TRANS_IO.IS_OPEN(TRANS_FILE)
      then
        TRANS_IO.CLOSE(TRANS_FILE);
      end if;
    if TRANS_IO.IS_OPEN(BANK_FILE)
      then
        TRANS_IO.CLOSE(BANK_FILE);
      end if;
    end REPORT_GEN;

```



## УПРАЖНЕНИЯ

1. Модифицируйте программу `SEQ_PROC_GRADES` из разд. 8.2.1 таким образом, чтобы она смогла обрабатывать переменное количество контрольных работ. При этом потребуются внести изменения в объявление комбинированного типа `BIG_REC`. Кроме того, положим, что названия предметов, по которым проводится контрольная работа, для одного и того же студента в каждом из двух входных файлов не обязательно расположены по алфавиту.

2. Модифицируйте программу `SEQ_PROC_GRADES` из разд. 8.2.1 таким образом, чтобы одно и то же значение строки `BIG_ST_ID` могло появляться в каждом входном файле более одного раза. Предполагается, что оба файла отсортированы.

3. Перепишите пакет `BANK_RESOURCES` из разд. 8.3.2, добавив в него подпрограммы, выполняющие добавление или изменение элементов типа `BANK_REC`. Перепишите программу `BANK_MAINT` с использованием нового пакета.

4. Возможно, что будет более эффективным иметь несколько элементов типа `TRANS_FILE_HEADER` в файле `TRANS_FILE`, используемом программой `CURR_TRANSACTION_PROC` из разд. 8.3.3. Перепишите программу `CURR_TRANSACTION_PROC` и пакет `TRANSACTION_RESOURCES` в предположении, что первые четыре элемента файла принадлежат к типу `TRANS_FILE_RECORD`. При этом каждый из первых четырех элементов файла содержит информацию примерно о восьми днях (точнее, о 8, 8, 8 и 7 днях).

5. Используя пакет `TRANSACTION_RESOURCES` из разд. 8.3.3, напишите программу, которая будет помечать выбранные сделки в файле `TRANS_FILE` как некорректные.

6. Перепишите процедуру `INSERT_REC` из программы `CURR_TRANSACTION_PROC` с использованием объявлений переименования так, чтобы глубина вложенности составных имен не превышала двух уровней.

7. Перепишите процедуру `POST_INDIV_BANK` из программы `POSTING_PROC` (см. разд. 8.3.3) так, чтобы она регистрировала сделку только в тех случаях, когда оба названия банков содержатся в файле `TRANS_FILE`.

8. Программа `POSTING_PROC` из разд. 8.3.3 регистрирует сделки без записи относительного положения сведений о сделке в файле `TRANS_FILE`. Внесите необходимые изменения в пакет `BANK_RESOURCES` и программу `POSTING_PROC` с тем, чтобы в информации о зарегистрированных сделках содержались данные об относительном положении сведений в файле `TRANS_FILE`.

9. В предположении, что модификации, предусмотренные в упр. 8, сделаны, перепишите программу `REPORT_GEN` из разд. 8.4, в которой теперь должны использоваться данные об относительном положении сведений о сделках в файле `TRANS_FILE`.

10. Напишите программу, которая выдает справку о тех банках, которые произвели более 20 сделок в месяце номер 07 (июль). Она должна генерировать текстовый файл с необходимыми заголовками.

# Структура программы и вопросы компиляции

## 9.1. СЕГМЕНТЫ КОМПИЛЯЦИИ И ПРОЦЕСС КОМПИЛЯЦИИ

### 9.1.1. Сегменты компиляции

Программы на Аде составляются из одного или более *сегментов компиляции*, хранящихся в библиотеке программ. Существуют два вида сегментов компиляции, которые имеют форму:

описание контекста	<u>Библиотечные сегменты</u>
описание контекста	<u>Вторичные сегменты</u>

В свою очередь *библиотечный сегмент*—это одна из следующих программных единиц:

- объявление или тело подпрограммы,
- объявление пакета,
- родовое объявление или его конкретизация.

*Вторичным сегментом* может быть:

- тело подпрограммы,
- тело пакета,
- подсегмент.

Мы уже пользовались каждым из перечисленных здесь сегментов компиляции, кроме подсегментов, которые будут описаны далее в настоящей главе.

*Описание контекста* указывает те библиотечные сегменты, имена которых необходимы внутри сегмента компиляции. Оно должно предшествовать библиотечным или вторичным сегментам. Пример описания контекста—это часто используемая строка:

```
with TEXT_IO; use TEXT_IO;
```

которая делает доступными подпрограммы (такие, как GET или PUT), необходимые для чтения и записи символов, строк и т.п. В следующем разделе даны примеры и приведены подробности описания контекста для сегментов компиляции.

Беглый взгляд на списки библиотечных и вторичных сегментов показывает, что в обоих списках имеются тела подпрограмм. Для тел подпрограмм справедлива следующая интерпретация: если библиотека программ уже содержит библиотечный сегмент, имеющий то же самое имя, что и тело, то это тело считается вторичным сегментом. В противном случае тело подпрограммы является одновременно и библиотечным, и вторичным сегментом.

### 9.1.2. Главные программы

В языке Ада не определяется, что составляет *главную программу*, т.е. программу, которая первой запускается средствами системного окружения, находящимися за пределами средств Ады. Точное определение требований к главной программе на языке Ада зависит от конкретной реализации языка. Однако в любой реализации процедура без формальных параметров может быть главной программой, а каждая

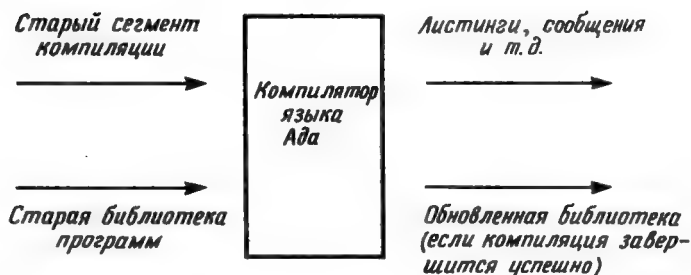


Рис. 9.1. Процесс компиляции.

главная программа должна быть подпрограммой, являющейся библиотечным сегментом. Ясно, что все программы на Аде, представленные ранее, могут быть названы главными программами, поскольку это – процедуры без формальных параметров.

### 9.1.3. Процесс компиляции

Сегменты компиляции, образующие программу на Аде, могут быть скомпилированы различными способами. Наиболее очевидный путь – представить для компиляции все сегменты в одном шаге. Но возможны и многие другие пути при отдельной компиляции сегментов. Вообще говоря, сегменты компиляции могут быть обработаны транслятором в любом порядке при условии выполнения определенных правил, необходимых для правильной генерации кода. Эти правила в основном выводятся путем последовательного применения правил видимости.

В процессе компиляции на вход транслятора подаются некоторые сегменты компиляции и модули из библиотечного файла. Если компиляция завершается успешно, то библиотечный файл обновляется. Это означает, что новая (только что оттранслированная) версия сегмента компиляции заменяет старые версии этого сегмента (если они имелись). Обобщенный процесс компиляции иллюстрирует рис. 9.1.

### 9.1.4. Описание контекста

Мы упоминали ранее, что описание контекста для сегмента компиляции необходимо в целях подключения к нему нужных библиотечных сегментов. Описание контекста выполняется с помощью фразы подключения контекста *with*:

with простое\_имя\_библиотечного\_сегмента;

В дополнение к этому присутствует фраза использования *use*, имеющая вид:

with простое\_имя\_библиотечного\_сегмента;

use простое\_имя\_библиотечного\_пакета;

В этой форме *use* нет ошибки. Таким образом, во фразе *with* разрешено указывать простое имя любого библиотечного сегмента (здесь могут быть имена подпрограмм, пакетов и родовых конкретизаций). Но во фразе *use* допускается указывать только имена пакетов. Заметьте также, что каждая фраза *with* создает зависимость между сегментами компиляции.

Пусть, например, имеется такой сегмент компиляции:

```

with PACK_A;
procedure PROC_B is
begin
  null;
end PROC_B;
  
```

Тогда `PACK_A` должен быть библиотечным сегментом и `PROC_B` зависит от `PACK_A`. Этот вид зависимости важен для установления допустимого порядка следования сегментов компиляции, а именно: *каждый библиотечный сегмент, от которого зависит данный сегмент компиляции, должен быть оттранслирован перед началом трансляции этого сегмента компиляции*. В нашем примере `PACK_A` должен быть оттранслирован перед тем, как может начаться трансляция `PROC_B`, потому что идентификаторы, принадлежащие `PACK_A`, видимы в `PROC_B`. Это правило является частным случаем более общего правила видимости: если идентификатор видим в программном сегменте, но не объявлен в нем, то этот идентификатор должен быть частью уже оттранслированного библиотечного сегмента.

В фразе `with` могут быть заданы имена нескольких библиотечных сегментов. В этом случае они должны быть разделены запятыми.

Всякий раз, когда какой-либо сегмент компиляции Ады передается на трансляцию, предполагается, что часть его контекста составляет пакет `STANDARD` (приведенный в приложении В). По этой причине типы `INTEGER`, `FLOAT`, `BOOLEAN`, `CHARACTER`, `STRING` и функции, выполняющие операции с ними, непосредственно видимы в любой программе на Аде. Пакет `STANDARD` содержит также пакет `ASCII` (состоящий из объявлений констант для управляющих символов и прочих специальных символов) и объявления некоторых предопределенных исключительных ситуаций (которые будут рассмотрены в гл. 11).

## 9.2. ПОДСЕГМЕНТЫ И ЗАГЛУШКИ

В начале главы в списке возможных вторичных сегментов были указаны и подсегменты. *Подсегменты* применяются при отдельной трансляции тел программных сегментов, объявленных внутри другого сегмента компиляции. Объявление программного сегмента внутри другого сегмента компиляции выполняется с помощью заглушки, которая может иметь три вида:

```
спецификация_подпрограммы is serapate
package body простое_имя_пакета is separate;
task body простое_имя_задачи is separate;
```

Последняя форма заглушки (для тела задачи) будет рассмотрена в следующей главе.

Зاغлушка занимает место соответствующего тела (т.е. фактического текста тела), которое задается в подсегменте. Подсегмент имеет вид:

```
separate (имя_порождающего_сегмента) соответствующее_тело;
```

Имя порождающего сегмента, указываемое в подсегменте, — это имя того сегмента компиляции, где употребляется заглушка (этот сегмент компиляции называется *порождающим сегментом*).

Преимущество использования заглушек состоит в том, что можно проектировать программу лишь в общих чертах, транслировать и реализовать только отдельные ее детали, а потом писать соответствующие тела сегментов (т.е. некоторый подробный текст). Этот подход известен под названием *нисходящего проектирования*. При использовании альтернативного метода разработки программ — *восходящего проектирования* — разработчик применяет уже существующие компоненты (возможно, имеющиеся в некоторых пакетах) как строительные блоки для создания программы, даже если эти компоненты могут не в полной мере соответствовать поставленной цели. Например, в предыдущих главах мы широко использовали пакет `CHECK_DATES_ALT` (и, в частности, процедуру `FILE_IN_DATE`) ввиду его возможностей преобразовывать дату, даже если его процедуры зачастую выполняли действия, ненужные для решения

поставленных задач. Несомненно, что оба подхода (и любой «гибридный» метод) одинаково хорошо подкрепляются средствами языка Ада.

**Пример.** Здесь иллюстрируется использование заглушек на примере иной формы записи объявлений для программы NAME\_PHONE из гл. 4.

```
with TEXT_IO; use TEXT_IO;
procedure SEP_NAME_PHONE is
-- Для переменных, начиная LINE_LN и кончая
-- DIGIT_CT, используйте такие же об'явления, как
-- и в программе NAME_PHONE из гл.4.
-- Далее располагаются заглушки.
procedure LOW_TO_UPPER_N_CT_COMMAS is separate;
procedure IGNORE_LEADING_SPACES is separate;
procedure FIND_NEXT_SP_OR_COMMA is separate;
procedure PLACE_SPACES is separate;
procedure IS_CORRECT_NAME is separate;
-- Все остальные процедуры также должны
-- компилироваться раздельно.
begin
-- Здесь используется такое же тело, как и в
-- программе NAME_PHONE.
end SEP_NAME_PHONE;
-- Соответствующие тела представлены следующими
-- подсегментами:
separate (SEP_NAME_PHONE)
procedure LOW_TO_UPPER_N_CT_COMMAS is
-- Следует точно такой же текст, как и в
-- исходной версии программы.
end LOW_TO_UPPER_N_CT_COMMAS;
-- Ниже располагается другой подсегмент.
separate (SEP_NAME_PHONE)
procedure IGNORE_LEADING_SPACES is
-- Следует точно такой же текст, как и в
-- исходной версии программы.
end IGNORE_LEADING_SPACES;
```

Весьма похожие подсегменты должны быть написаны и для остальных подпрограмм.

**Пример.** В этом примере используются заглушки и подсегменты из программы ACCR\_INTEREST в гл. 5. В новой версии вначале дается текст порождающего сегмента.

```
procedure SEP_ACCR_INTEREST is
-- Используйте точно такие же об'явления типов и
-- об'ектов, как и в программе ACCR_INTEREST.
function IS_VALID_DATE (FORM_DATE : DATE)
    return BOOLEAN is separate;
procedure FILL_IN_DATE (PROC_F_DATE : in out DATE;
    GOOD_DATE : out BOOLEAN)
    is separate;
function FIND_COUP_DATE (FORM_MAT_DATE,
    FORM_SETL_DATE : DATE)
    return DATE is separate;
begin
-- Тело программы ACCR_INTEREST не меняется.
end SEP_ACCR_INTEREST
-- Подсегменты, соответствующие приведенным выше
-- заглушкам, таковы:
```

```

separate (SEP_ACCR_INTEREST)
function IS_VALID_DATE (FORM_DATE : DATE)
    return BOOLEAN is
-- Здесь размещается тело подпрограммы.
end IS_VALID_DATE;
-- Ниже располагается другой подсегмент.
separate (SEP_ACCR_INTEREST)
procedure FILL_IN_DATE ( PROC_F_DATE : in out DATE;
                        GOOD_DATE : out BOOLEAN )
    is
-- Здесь размещается тело подпрограммы.
end FILL_IN_DATE;
separate ( SEP_ACCR_INTEREST )
function FIND_COUP_DATE ( FORM_MAT_DATE,
                        FORM_SETL_DATE : DATE )
    return DATE is
-- Здесь размещается тело подпрограммы.
end FIND_COUP_DATE;

```

**Пример.** В заключение приводится пример заглушки и соответствующего подсегмента для случая пакетов. Здесь используется программа `BANK_MAINT` из гл. 8.

```

with TEXT_IO; use TEXT_IO;
-- Остальная информация о контексте, кроме инфор-
-- мации о пакете BANK_RESOURCES, остается без
-- изменений.
procedure SEP_BANK_MAINT is
-- Здесь внесено изменение: вместо строки
-- with BANK_RESOURCES; use BANK_RESOURCES;
-- записываются следующие строки:
package BANK_RESOURCES is
-- Копия спецификации пакета.
end BANK_RESOURCES;
-- Далее располагается заглушка.
package body BANK_RESOURCES is separate;
-- Все остальное в этой программе совпадает с
-- текстом исходной программы.
end SEP_BANK_MAINT;

```

Соответствующий подсегмент, порождающим сегментом для которого является `SEP_BANK_MAINT`, имеет вид

```

separate (SEP_BANK_MAINT)
package body BANK_RESOURCES is
-- Здесь размещается копия тела одноименного
-- пакета из гл. 8.
end BANK_RESOURCES;

```

Основное правило компиляции для подсегментов заключается в том, что трансляция порождающего сегмента должна производиться перед трансляцией нужного тела, размещенного в соответствующем подсегменте. Например, перед компиляцией подсегмента, содержащего тело пакета `BANK_RESOURCES` (см. предыдущий пример), программа `SEP_BANK_MAINT` уже должна быть оттранслирована. Очевидно, что это правило отражает другой вид зависимости, в какой-то степени дополняющей зависимость, вносимую фразами `with`.

Компиляция подсегмента (скажем, тела пакета `BANK_RESOURCES`) выполняется в контексте порождающего сегмента (в данном случае `SEP_BANK_MAINT`). Если,

вдобавок, подсегмент имеет свое описание контекста, то при компиляции предполагается, что оно добавляется к описанию контекста порождающего сегмента. Это правило может быть применено на более высоком уровне повторно, если порождающий сегмент сам является подсегментом.

### 9.3. ПРАВИЛА КОМПИЛЯЦИИ И ПЕРЕКОМПИЛЯЦИИ

Итак, сегменты компиляции Ады могут быть оттранслированы в любом порядке при условии соблюдения двух следующих правил, которым должен подчиняться этот порядок:

1. Вторичный сегмент должен быть оттранслирован после трансляции соответствующего библиотечного сегмента. Поэтому тела пакетов, тела подпрограмм и подсегменты должны компилироваться после соответствующих спецификаций пакетов, спецификаций подпрограмм или порождающих сегментов. Но необходимо помнить, что тело не родовой подпрограммы может служить своей собственной спецификацией.

2. Сегмент компиляции должен быть оттранслирован после того, как будут оттранслированы все библиотечные сегменты, упоминаемые в его операторах описания контекста. Если во время трансляции сегмента возникает ошибка, то библиотека программ не модифицируется.

В Аде есть также два правила перекомпиляции, согласующиеся с приведенными выше правилами компиляции:

1. Ввиду того что вторичные сегменты должны пройти трансляцию после соответствующих им библиотечных сегментов, любая перекомпиляция библиотечного сегмента делает связанные с ним вторичные сегменты некорректными, и поэтому их следует перетранслировать. Однако любая перекомпиляция вторичного сегмента не делает непригодным связанный с ним библиотечный сегмент, и поэтому нет нужды в какой-либо повторной трансляции библиотечного сегмента.

2. Если перетранслируется какой-либо сегмент компиляции из описания контекста, то выходит из употребления тот сегмент компиляции, к которому относится этот контекст, и его следует перекомпилировать.

Для иллюстрации правил компиляции рассмотрим программу `CURR_TRANSACTION_PROC` из разд. 8.3. В описании контекста дан список таких пакетов, как `CHECK_DATES_ALT`, `BANK_RESOURCES`, `TRANSACTION_RESOURCES`, `TEXT_IO` и `LEGAL_HOLIDAYS`. Можно было бы просто передать на трансляцию эти пакеты (их спецификации и тела) и саму программу в виде одного большого сегмента компиляции. Можно передать на трансляцию и несколько отдельных сегментов компиляции, таких, как `CHECK_DATES_ALT`, `BANK_RESOURCES` и `TRANSACTION_RESOURCES`, в любом порядке, поскольку все эти пакеты независимы друг от друга. Потом можно оттранслировать пакет `LEGAL_HOLIDAYS`, так как он зависит от `CHECK_DATES_ALT`. В заключение можно оттранслировать программу `CURR_TRANSACTION_PROC`. Возможно множество вариаций этого порядка. Например, можно подавать на вход транслятора по два сегмента компиляции сразу, скажем `BANK_RESOURCES` и `TRANSACTION_RESOURCES`. Независимо от порядка трансляции заданной программы, если правила компиляции будут соблюдаться, библиотечный файл будет обновлен единообразным способом.

Как упоминалось выше, во время компиляции на вход транслятора поступают один или более сегментов компиляции и модули из библиотеки программ. Для каждой компиляции предполагается наличие единственной библиотеки программ, несмотря на то что в конкретной реализации Ады может иметься несколько библиотек программ. Следует ожидать, что на конкретной вычислительной установке будут иметься и некоторые другие особенности реализации, касающиеся работы с библиотечным

файлом, такие, как создание библиотечного файла, объединение библиотечных файлов и выдача запросов по состоянию библиотечных или вторичных сегментов в библиотечном файле. Также отметьте, что обработка библиотечных сегментов во время выполнения программы выполняется способом, согласующимся с приведенными здесь правилами компиляции.

## УПРАЖНЕНИЯ

1. Представьте, что программа NAME\_PHONE в гл. 4 написана с помощью нисходящего метода. Перепишите эту программу, применяя заглушки и подсегменты для каждой из вызываемых подпрограмм.
2. Выполните упр. 1 для программы ACCR\_INTEREST из гл. 5.



# Задачи

### 10.1. ЗАДАЧИ И МЕХАНИЗМ РАНДЕВУ

Все представленные до этого момента программы на Аде выполнялись последовательно, т. е. никакие два оператора не выполнялись одновременно. Зачастую, однако, решаемая практическая задача может быть лучше представлена в виде параллельного выполнения двух или более действий. Например, банковские операции часто производятся несколькими кассирами одновременно; выполняется управление сразу несколькими самолетами, производящими в одно и то же время посадку или взлет; телефонная компания должна иметь дело с несколькими телефонными разговорами, ведущимися параллельно. Хотя каждый из этих примеров может быть смоделирован при помощи последовательно выполняемых операторов, более естественным подходом будет придание языку программирования средств, позволяющих работать с параллельными процессами.

#### 10.1.1. Задачи

Работа с параллельными процессами в языке Ада осуществляется с помощью средств, называемых задачами, выполнение которых происходит параллельно. *Задачи* являются программными модулями языка Ада. Это — последний вид модулей, с которыми мы познакомимся после подпрограмм, пакетов и родовых модулей. Разные задачи могут работать независимо, хотя они и имеют средства для связи друг с другом. Ада не регламентирует, как должны быть реализованы параллельные задачи, т. е. то, сколько компьютеров (но не менее одного) следует использовать, или какой конкретный вид выполнения нужно применить. Однако Ада интерпретирует задачи как логические объекты, ведущие себя так, как будто бы они выполнялись параллельно на разных машинах.

Задачи, как подпрограммы и пакеты, имеют две части: спецификацию и тело. Как и в случае подпрограмм и пакетов, спецификация задачи с последующим символом «;» составляет объявление задачи. В противоположность подпрограммам и пакетам задачи не могут быть оттранслированы самостоятельно. Поэтому для компиляции задачи должны быть помещены в подпрограмму или пакет.

#### 10.1.2. Рандеву

Задачи языка Ада связываются между собой при помощи входов. Если одна задача выдала обращение ко входу и оно принято другой задачей, то обе задачи теряют свою независимость: они устанавливают *рандеву*, и до тех пор, пока рандеву действует, задачи синхронизированы.

Эти понятия будут вскоре проиллюстрированы, но сейчас запомните, что в механизме рандеву отсутствует симметрия. То есть одна задача может инициировать возможность установления рандеву, обратившись ко входу, а другая задача может принять, а может и не принимать вызов, выданный первой задачей. Рандеву происхо-

дит только тогда, когда вторая задача принимает вызов. Рандеву может быть в конце концов прекращено, и тогда выполнение каждой задачи может продолжаться независимо.

### 10.1.3. Спецификация задачи

Спецификация задачи может определять единственную задачу или тип объектов «задача». Тип «задача» распознается по зарезервированному слову `type`, появляющемуся в ее спецификации. Наиболее простая форма спецификации, определяющая единственную задачу:

```
task идентификатор_задачи
```

Если после члена «идентификатор\_задачи» поставить символ «;», то получится объявление задачи. Например, можно было бы объявить задачу так:

```
task CHECK_BANKS_FOR_OVERFLOW;
```

### 10.1.4. Входы

Для того чтобы к данной задаче могли обращаться другие задачи, необходимо указать точки входов. Точки входов, если они есть, должны быть описаны в спецификации задачи при помощи *объявления входа*. В этом случае спецификация имеет вид:

```
task идентификатор_задачи is  
  entry идентификатор_входа (дискретный_диапазон) формальная_часть;  
  -- Далее можно поместить другие объявления входов или  
  -- фразы представления (см. гл. 7).
```

```
end идентификатор_задачи;
```

Дискретный диапазон (используемый, например, в регулярных типах) и формальная часть (используемая, например, в спецификациях подпрограмм) необязательны.

**Пример спецификации задачи:**

```
task MAINT_TRAN_FILES is  
  entry LOOK_FOR_FILES ( 1 .. 12 )  
    (FORM_MO : INTEGER;  
     FORM_DAY : INTEGER);  
end MAINT_TRAN_FILES;
```

В этом объявлении задачи при описании входа задачи дискретный диапазон (12 элементов для 12 возможных месяцев) и формальная часть с двумя формальными параметрами типа `INTEGER`. Другой пример:

```
task LOOKING_FOR_BANKS is  
  entry INQUIRY  
    (INQ_BANK_NAME : in out STRING;  
     INQ_POS       : out POSITIVE_COUNT;  
     INQ_FOUND     : out BOOLEAN) is  
end LOOKING_FOR_BANKS;
```

Вход `INQUIRY` имеет формальную часть. Он выполняет проверку, чтобы определить, находится ли уже название банка в файле `BANK_FILE`. Если банк найден, то вход возвращает ответ `TRUE` через переменную `INQ_FOUND`. Это похоже на процедуру `RETRIEVE` пакета `BANK_RESOURCES` из гл. 8 с той лишь разницей, что теперь могут выполняться одновременно несколько параллельных запросов.

Заметьте, что объявления входов имеют вид, похожий на спецификацию подпрограмм. Сходство заходит еще дальше: вход задачи может быть вызван в тех же случаях, когда разрешен вызов подпрограммы, а правила согласования фактических и формальных параметров одинаковы и для задач, и для подпрограмм.

Однако для задач не существует понятия рекурсивного вызова. А вызов задачей самой себя, прямо либо косвенно, является ошибкой, как показано в разд. 10.1.10.

Формальные параметры, входящие в состав формальной части объявления входа, могут, как показывает предыдущий пример, иметь вид связи *in* (только для чтения), *in out* (и для чтения, и для записи) или *out* (только для записи).

Обращения ко входам из вызывающей задачи также имеют форму вызова процедур: за именем входа может следовать список фактических параметров, заключенный в скобки, после которых стоит символ «;». Префикс, состоящий из имени задачи, обязателен при обращении к ней, поскольку для задач недопустима фраза *use* (вспомните, что задачи не являются сегментами компиляции).

Возможно, что несколько задач будут обращаться к одному и тому же входу, в этом случае только одной задаче в данный момент времени будет разрешено рандеву. Остальные задачи будут ожидать в очереди, которая должна быть обеспечена в любой реализации Ады. Задачи, ожидающие в очереди, будут приняты вызываемой задачей по принципу: первым пришел – первым обслужен. О тех задачах, которые находятся в очереди, говорят, что они *приостановлены*. Если две или более задачи обращаются к одной и той же задаче одновременно, то порядок их обслуживания не определен.

Вот пример обращения ко входу:

```
LOOKING_FOR_BANKS.INQUIRY (THIS_BANK, TNIS_POS, IS_THERE);
```

Этот вызов входа (в предположении, что названная задача видима) будет пытаться установить рандеву с *LOOKING\_FOR\_BANKS*. Если немедленное рандеву невозможно, то вызов будет поставлен в очередь, предусмотренную для входа *INQUIRY*.

### 10.1.5. Тело задачи

Как упоминалось ранее, типы «задача» описываются точно так же, как и отдельные задачи, за исключением того, что при спецификации типов за зарезервированным словом *task* следует зарезервированное слово *type*. Разумеется, после объявления типа «задача» можно описывать объекты этого типа. Использование объектов типа «задача» весьма ограничено, поскольку типы «задача» относятся к ограниченным приватным типам: их нельзя проверять на равенство или неравенство и их значения нельзя присваивать переменным.

Тело задачи имеет вид

```
task body идентификатор_задачи is
```

-- Здесь размещается декларативная часть.

```
begin
```

-- Здесь располагается последовательность операторов.

```
end идентификатор_задачи;
```

После члена «последовательность операторов» может располагаться необязательная часть для обработки исключительных ситуаций, которая будет описана в гл. 11.

### 10.1.6. Оператор приема асерт

Если в спецификации задачи имеется объявление входа, то тело задачи должно содержать по крайней мере один *оператор приема* (т.е. *оператор асерт*). Приблизительный смысл этого таков: спецификация показывает, может ли задача быть вызвана

другой задачей (это достигается с помощью объявлений входов), и определяет вид должного обращения к ней. Фактические условия, при которых наступает рандеву, указываются в соответствующих операторах ассерт (для каждого объявления входа может иметься более одного соответствующего ему оператора *assert*).

Простейшая форма оператора ассерт такова:

assert идентификатор\_входа;

Эта форма применяется, если соответствующее объявление входа не имеет ни описателя дискретного диапазона, ни формальной части (они, как отмечалось ранее, необязательны для объявления входа). Если же некоторые из этих необязательных частей присутствуют в объявлении входа, то и соответствующий оператор ассерт также должен иметь их. Например, оператор ассерт имеет вид:

assert идентификатор\_входа (выражение) формальная\_часть;

Полная форма оператора ассерт может содержать член «последовательность\_операторов», заключенный между зарезервированными словами *do ... end*, после которых ставится «;». Эта последовательность операторов выполняется во время рандеву. Полная форма оператора такова:

assert идентификатор\_входа (выражение) формальная\_часть  
do последовательность\_операторов end;

Как упоминалось ранее, рандеву вызывающей и вызываемой задач начинается с согласования фактических и формальных параметров. Затем оно продолжается выполнением операторов (если они есть), расположенных между *do* и *end*, и при достижении символа «;» рандеву заканчивается. Во время рандеву задачи могут обмениваться информацией (через список параметров) и, кроме того, они синхронизированы: член «последовательность\_операторов» выполняется «от имени» обеих задач<sup>1)</sup>.

### 10.1.7. Владельцы

Мы констатировали ранее, что задачи не являются сегментами компиляции и поэтому не могут быть самостоятельно оттранслированы. Обычно их помещают в описательную часть пакета или подпрограммы. Поэтому задача должна находиться в зависимости от некоторого «владельца», который может представлять из себя подпрограмму, блок, пакет или другую задачу. Если задача появляется в декларативной части, к примеру, блока, то этот блок будет владельцем данной задачи. Задача может иметь нескольких владельцев. Например, если блок, содержащий задачу, запускается другой задачей, то вторая задача, как и блок, будет являться владельцем первой задачи. Поэтому ясно, что задача может зависеть от нескольких программных модулей или блоков. Понятие владельца важно, как будет вскоре показано, для установления условий завершения задачи.

### 10.1.8. Примеры и программа

Данный пример иллюстрирует то, как можно написать тело задачи *LOOKING\_FOR BANKS* для ее спецификации, приведенной ранее.

<sup>1)</sup> Эта последовательность операторов в литературе называется критической секцией, и на время ее выполнения вызывающая задача приостанавливается. — *Прим. перев.*

```

task body LOOKING_FOR_BANKS is
  NO_FILES_PROCESSED : NATURAL := 0;
  -- Предполагается, что об'явления пакета
  -- BANK_RESOURCES видимы, а сам пакет
  -- обработан.
begin
  loop
    accept INQUIRY
      (INQ_BANK_NAME : in out STRING;
       INQ_POS       : out POSITIVE_COUNT;
       INQ_FOUND     : out BOOLEAN)
    do
      -- Следующие операторы будут вы-
      -- полняться во время randevу.
      RETRIEVE(FORM_BANK_NAME => INQ_BANK_NAME,
               FORM_POS       => INQ_POS,
               FORM_FOUND     => INQ_FOUND );
      if INQ_BANK_NAME = "12345678901234567890"
      then
        NO_FILES_PROCESSED :=
          NO_FILES_PROCESSED + 1;
      end if;
      -- Рандеву заканчивается при выполнении
      -- следующего оператора.
    end INQUIRY;
    exit when NO_FILES_PROCESSED = 2;
    -- Когда количество закрытых файлов
    -- становится равным 2, то дополни-
    -- тельных входных данных ожидать
    -- не следует.
  end loop;
  CLOSE (BANK_FILE);
end LOOKING_FOR_BANKS;

```

Эти спецификацию и тело LOOKING\_FOR\_BANKS можно поместить в новый пакет, названный PARALLEL\_BANK\_RESOURCES и имеющий такую спецификацию:

```

with BANK_RESOURCES; use BANK_RESOURCES;
package PARALLEL_BANK_RESOURCES is
  -- Здесь располагается спецификация задачи
  -- LOOKING_FOR_BANKS.
end PARALLEL_BANK_RESOURCES;
--
package body PARALLEL_BANK_RESOURCES is
  -- Здесь располагается тело задачи
  -- LOOKING_FOR_BANKS.
begin
end PARALLEL_BANK_RESOURCES;

```

В программе, приведенной ниже, используется задача из пакета PARALLEL\_BANK\_RESOURCES и еще две задачи. Эта программа считывает названия банков из двух последовательных файлов и проверяет, есть ли эти имена в файле BANK\_FILE. После того как для обоих последовательных файлов будет достигнуто состояние END\_OF\_FILE (Конец файла), программа выдает отношение количества банков, найденных в файле BANK\_FILE, к общему числу банков для каждого из последовательных файлов и завершается.

## Программа GATHER\_BANK\_STATISTICS

```

with TEXT_IO; use TEXT_IO;
with SEQUENTIAL_IO, DIRECT_IO,
    PARALLEL_BANK_RESOURCES;
use DIRECT_IO, PARALLEL_BANK_RESOURCES;
procedure GATHER_BANK_STATISTICS is
    package SEQ_1 is new SEQUENTIAL_IO(STRING(1..20));
    package SEQ_2 is new SEQUENTIAL_IO(STRING(1..20));
    use SEQ_1, SEQ_2;
    package INT_IO is new INTEGER_IO(INTEGER);
    use INT_IO;
    FILE_1 : SEQ_1.FILE_TYPE;
    FILE_2 : SEQ_2.FILE_TYPE;
    -- Далее располагаются спецификации двух задач --
    -- CHECK_FILE_1 и CHECK_FILE_2. Эти задачи не имеют
    -- входов.
    --
    task CHECK_FILE_1;
    task CHECK_FILE_2;
    --
    task body CHECK_FILE_1 is
        NO_HITS_FILE_1, TOT_FILE_1 : NATURAL := 0;
        BNK_1 : STRING (1 .. 20);
        BNK_1_POS : NATURAL;
        IS_1_THERE : BOOLEAN;
        FILE_1 : FILE_TYPE;
    begin
        OPEN (FILE => FILE_1,
            MODE => IN_FILE,
            NAME => "FILE1.DAT"
            FORM => "");
        while not END_OF_FILE (FILE_1)
            loop
                READ (FILE => FILE_1, ITEM => BNK_1);
                TOT_FILE_1 := TOT_FILE_1 + 1;
                LOOKING_FOR_BANKS.INQUIRY
                    (BNK_1, BNK_1_POS, IS_1_THERE);
                -- В этом операторе выполняется вызов входа
                -- задачи LOOKING_FOR_BANKS. Вызываемая задача
                -- будет приостановлена до завершения рандеву.
                -- После начала рандеву вызываемая задача бу-
                -- дет синхронизирована с вызываемой, и эти
                -- задачи станут обмениваться информацией.
                -- Например, переменная IS_1_THERE будет нести
                -- информацию о том, содержится ли в файле
                -- BANK_FILE название банка, задаваемое фак-
                -- тическим параметром. Начиная со следующего
                -- оператора, обе задачи станут вновь незави-
                -- симыми друг от друга, и это будет продолжа-
                -- ться вплоть до следующего рандеву.
                if IS_1_THERE
                    then
                        NO_HITS_FILE_1 := NO_HITS_FILE_1 + 1;
                    end if;
            end loop;
        end loop;

```

```

LOOKING_FOR_BANKS.INQUIRY("12345678901234567890",
    BNK_1_POS, IS_1_THERE );
-- Этот вызов входа призван гарантировать то,
-- что данная задача в конце концов завершится.
PUT(" The number of banks in FILE1.DAT is ");
PUT( TOT_FILE_1 );
PUT(" The number of banks in FILE1.DAT " &
    "and BANK_FILE is ");
PUT( NO_HITS_FILE_1 );
CLOSE (FILE_1);
end CHECK_FILE_1;
--
-- Эта задача является копией задачи CHECK_FILE_2,
-- использующей второй последовательный файл.
-- В упр.1 в конце главы требуется применить
-- иной подход к реализации программы.
--
task body CHECK_FILE_2 is
    NO_HITS_FILE_2, TOT_FILE_2 : NATURAL := 0;
    BNK_2 : STRING (1 .. 20);
    BNK_2_POS : NATURAL;
    IS_2_THERE : BOOLEAN;
    FILE_2 : FILE_TYPE;
begin
    OPEN (FILE => FILE_2,
        MODE => IN_FILE,
        NAME => "FILE2.DAT"
        FORM => "");
    while not END_OF_FILE (FILE_2)
    loop
        READ (FILE => FILE_2, ITEM => BNK_2);
        TOT_FILE_2 := TOT_FILE_2 + 1;
        LOOKING_FOR_BANKS.INQUIRY
            (BNK_2, BNK_2_POS, IS_2_THERE);
        if IS_2_THERE
        then
            NO_HITS_FILE_2 := NO_HITS_FILE_2 + 1;
        end if;
    end loop;
    LOOKING_FOR_BANKS.INQUIRY("12345678901234567890"
        , BNK_2_POS, IS_2_THERE );
    PUT(" The number of banks in FILE2.DAT IS ");
    PUT( TOT_FILE_2 );
    PUT(" The number of banks in FILE2.DAT " &
        "and BANK_FILE is ");
    PUT( NO_HITS_FILE_2 );
    CLOSE( FILE_2 );
end CHECK_FILE_2;
--
begin
-- Здесь начинается главная программа. В ней
-- должен быть, по крайней мере хотя бы
-- один пустой оператор.
null;
end GATHER_BANK_STATISTICS;

```

### 10.1.9. Состояния задач

Задачи становятся *активными* — т.е. они начинают свою логически независимую работу, действуя так, как будто бы они выполнялись на разных машинах, — когда в пакете или подпрограмме, которые их содержат, встретится зарезервированное слово `begin`, помечающее начало члена «последовательность операторов». Запуск задачи означает последовательное выполнение операторов из исполняемой части ее тела.

Задача заканчивается, когда достигается конец ее тела. Понятие окончания выполнения, как это демонстрируют приведенные ранее программы, справедливо также для подпрограмм и блоков. Окончание задачи, подпрограммы или блока может произойти и по многим другим причинам, например, при возникновении исключительных ситуаций. Это будет рассмотрено в следующей главе.

В предыдущей программе имеются три задачи (кроме главной программы, которая также считается задачей). Владелец задач `CHECK_FILE_1` и `CHECK_FILE_2` является процедура `GATHER_BANK_STATISTICS`. Владелец третьей задачи `LOOKING_FOR_BANKS` — это пакет `PARALLEL_BANK_RESOURCES`. Эти задачи могут находиться в одном из трех состояний: выполняются (т.е. выполняются их операторы), приостановлены (например, в ожидании randevu) или полностью завершены (например, закончилось выполнение данной задачи и всех других зависящих от нее задач). Полное завершение задачи, которое будет детально рассмотрено в следующем разделе, — это более общее понятие, чем окончание выполнения задачи. «Окончание выполнения» означает лишь то, что при последовательном выполнении операторов задачи достигнуто зарезервированное слово `end`. Окончание выполнения данной задачи никак не связано с работой зависящих от нее задач, которые могут продолжать свои действия. Напротив, понятие «полного завершения» подразумевает окончание выполнения этой задачи и, возможно, полное завершение зависящих от нее задач и ее задач-владельцев.

Обратимся опять к предыдущей программе. Если задача `LOOKING_FOR_BANKS` достигнет оператора `assert`, то может случиться так, что ни одна из других задач еще не обратилась к ней. В этом случае выполнение `LOOKING_FOR_BANKS` будет приостановлено и она будет находиться в этом состоянии до тех пор, пока не станет возможным randevu. Фактически, если в будущем так и не произойдет обращения к этой задаче, она никогда не выйдет из приостановленного состояния. Для предотвращения такой ситуации мы ввели в программу счетчик количества закрытых входных файлов, и поэтому по окончании обработки всех входных файлов производится выход из цикла.

Возможно, что все три задачи будут выполняться одновременно (когда они запускаются в первый раз, то каждая задача независимо от других открывает файл), но может случиться и так, что одна или две задачи будут в данный момент приостановлены. Например, приостанавливаются сразу две задачи, если в одной из них выполняется оператор `assert`, а другая в это время уже приостановлена в ожидании randevu. В этом случае одна из задач ожидает randevu в очереди, связанной с данным оператором `assert`. В предыдущей программе в состоянии ожидания никогда не могло находиться более одной задачи, поскольку каждая из задач, считывающих входной последовательный файл, после каждой операции чтения приостанавливается до завершения ее randevu с `LOOKING_FOR_BANKS`. Такая ситуация возникает потому, что здесь отсутствует буферная задача, а вызывающая задача ожидает завершения своего randevu. Вероятно, затрачивается большее время на проверку и обработку нескольких записей из файла `BANK_FILE`, чем на чтение из последовательного файла.

Теперь подытожим некоторые свойства асимметрии механизма randevu:

- вызываемая задача «не знает», какая из задач обращается к ней (а вызывающая задача «знает», куда она обращается). Так, задаче `LOOKING_FOR_BANKS` не известно, имеет ли она randevu с `CHECK_FILE_1` или же с `CHECK_FILE_2`;



CHECK\_FILE\_1 LOOKING\_FOR\_BANKS CHECK\_FILE\_2

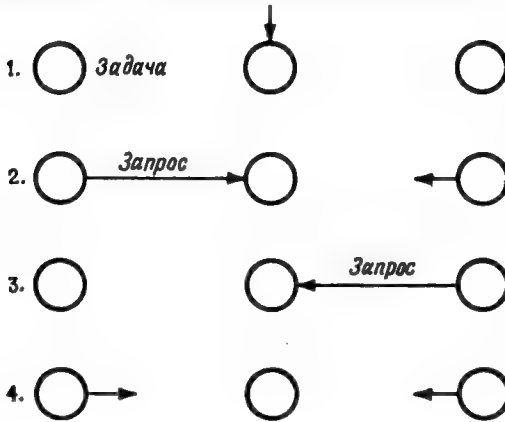


Рис. 10.1. Механизм рандеву.

Момент времени:

1-задача LOOKING\_FOR\_BANKS ожидает вызова. Другие задачи не выдают вызовов;  
 2-происходит рандеву между задачами CHECK\_FILE\_1 и LOOKING\_FOR\_BANKS. Задача CHECK\_FILE\_2 ожидает своей очереди;  
 3-рандеву между задачами CHECK\_FILE\_2 и LOOKING\_FOR\_BANKS, 4-задачи CHECK\_FILE\_1 и CHECK\_FILE\_2 пытаются вызвать задачу LOOKING\_FOR\_BANKS (которая не готова).

– вызываемая задача приостанавливает вызывающую на время действия рандеву.

Заметьте, что во время рандеву вызываемая задача может выдать запрос на проведение рандеву с другой задачей. В этой ситуации новое рандеву должно закончиться раньше, чем может завершиться предыдущее.

Рис. 10.1 иллюстрирует различные состояния для трех задач из программы GATHER\_BANK\_STATISTICS при разных условиях.

### 10.1.10. Блокировка

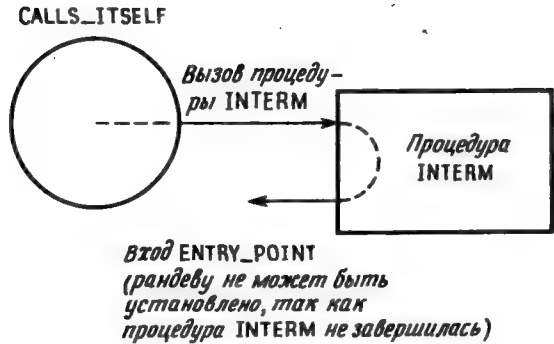
Необходимо проявлять особую осторожность, чтобы избежать такого положения, когда все задачи в программе приостанавливаются в ожидании выполнения друг друга. Эта ситуация называется *блокировкой*. Ниже приведен простой пример блокировки.

**Пример.** Рассмотрим следующую процедуру:

```
procedure DEADLOCK_EXAMPLE is
  task CALLS_ITSELF is
    entry ENTRY_POINT;
  end CALLS_ITSELF;
  procedure INTERM is
    CALLS_ITSELF.ENTRY_POINT;
  end INTERM;
  task body CALLS_ITSELF is
    INTERM;
    accept ENTRY_POINT;
  end CALLS_ITSELF;
begin
  null;
end DEADLOCK_EXAMPLE;
```

В этой программе запускается задача CALLS\_ITSELF, затем она вызывает процедуру INTERM, которая в свою очередь выдает обращение ко входу ENTRY\_POINT. Ввиду того что оператор ассерт не достигается, вызов приостанавливается. Но оператор ассерт никогда не будет выполнен, так как вначале должно закончиться выполнение предшествующего оператора. Этот пример блокировки иллюстрирует рис. 10.2.

Рис. 10.2. Взаимная блокировка.



Распознать возможность возникновения блокировки и избежать ее весьма непросто. В реальных ситуациях причины блокировки могут быть в противоположность данному примеру далеко не так очевидны.

## 10.2. ОПЕРАТОРЫ И АТТРИБУТЫ ДЛЯ ЗАДАЧ

### 10.2.1. Оператор задержки

Выполнение задачи может быть временно приостановлено с помощью *оператора задержки* *delay*. Его форма такова:

*delay* выражение\_для\_длительности;

Член «выражение\_для\_длительности» принадлежит к предопределенному фиксированному типу *DURATION*. Описание этого типа входит в пакет *STANDARD*, текст которого дан в приложении В. Оператор *delay* приостанавливает выполнение задачи по меньшей мере на то количество секунд, которое задано в «выражении\_для\_длительности».

Предопределенный пакет *CALENDAR*, текст которого приведен в приложении В, определяет операции, возможные для величин приватного типа *TIME*. Вот некоторые из перекрывающихся операций, определенные в этом пакете: *+*, *-*, *<*, *<=*, *>* и *>=*. Эти функции возвращают значения типов *TIME* или *DURATION* и имеют формальные параметры типов *TIME* или *DURATION*.

Пакет *CALENDAR* содержит и некоторые другие полезные подпрограммы, такие, как *SPLIT* (по заданному значению времени она возвращает значения *YEAR*—год, *MONTH*—месяц, *DAY*—число и *SECONDS*—секунды) и *TIME\_OF* (которая выполняет обратные по сравнению со *SPLIT* действия: по заданным значениям типов *YEAR*, *MONTH*, *DAY* и *SECONDS* возвращает значение типа *TIME*). Есть в этом пакете и ряд других функций, которые независимо выполняют операции над величинами некоторых из перечисленных типов. Пакет *CALENDAR* используется в программе *PLANT\_SCHED*, которая будет приведена далее.

### 10.2.2. Оператор отбора. Выборочное ожидание

Существуют три вида *операторов отбора* (*select*), имеющие названия: выборочное ожидание, условный вызов входа и таймированный вызов входа. Обсудим их по очереди.

Оператор *выборочного ожидания* имеет форму:

```

select;
альтернатива_отбора
--- В операторе select может быть одна или несколько
--- альтернатив отбора.
or
альтернатива_отбора
else
последовательность_операторов
end select;

```

Здесь допустимы одна или несколько альтернатив отбора, разделяемых зарезервированным словом `or`. Альтернативы могут отсутствовать. Часть `else` необязательна.

Альтернатива отбора может иметь следующие три разновидности:

```

when условие => оператор_приема
                  последовательность_операторов
when условие => оператор_задержки
                  последовательность_операторов
when условие => terminate;

```

Необязательными частями здесь являются: «последовательность\_операторов» и «when условие =>» (условие отбора альтернативы).

Если для альтернативы отсутствует зарезервированное слово `when` или же если условие, следующее за словом `when`, истинно, то соответствующая альтернатива оператора `select` называется *открытой*. В противном случае она называется *закрытой*. Эти термины полезны для понимания работы операторов селективного ожидания.

В операторе селективного ожидания должна иметься по крайней мере одна альтернатива приема с оператором `assert`. Выполнение оператора селективного ожидания происходит следующим образом. Вначале в некотором произвольном порядке вычисляются условия, стоящие за зарезервированными словами `when`. Затем одна из альтернатив отбора или `else`-часть, если только она имеется, будет претендовать на выполнение. Предпочтение будет отдано любой из открытых альтернатив с оператором приема, которая может установить рандеву. Если несколько альтернатив удовлетворяет этому требованию, то одна из них будет выбрана случайным образом. Выбор альтернативы с оператором приема означает, что будут выполнены и последовательность операторов, входящая в оператор приема (если она есть), и последовательность операторов, следующая за оператором приема (если эта последовательность имеется).

Если ни для какой альтернативы приема невозможно осуществить рандеву и если `else`-часть отсутствует, то задача войдет в состояние ожидания. Однако если имеется `else`-часть, то она будет отображена и выполнена.

В то время как задача будет находиться в состоянии ожидания, будет выполняться отбор открытой альтернативы задержки с оператором `delay`. Это произойдет, если заданная задержка истекла, а никакая альтернатива приема не могла быть отображена. Если в нескольких альтернативах задержки указана одинаковая длительность, то будет отображена случайным образом одна из этих альтернатив.

В конце концов будет отображена терминирующая альтернатива, если все зависимые задачи данной задачи-владельца завершились или же находятся в состоянии ожидания по терминирующей альтернативе, а в очередях ко входам этих задач нет ни одного вызова. Если все альтернативы закрыты, а `else`-часть отсутствует, то возникает исключительная ситуация `PROGRAM_ERROR`. Пояснения по этой ситуации будут даны в гл. 11.

Обратите внимание, что отбор терминирующей альтернативы в операторе селек-



```

if CHANGE_BANK_NAME = "12345678901234567890"
then
  NO_FILES_PROCESSED :=
    NO_FILES_PROCESSED + 1;
else
  -- Если банк найден, то запись обновляется.
  if CHANGE_FOUND
  then
    CHANGE_BANK_REC.MAST_HEAD.BANK_NAME :=
      FORM_BANK_NAME;
    CHANGE_BANK_REC.MAST_HEAD.OTHER_INFO :=
      CHANGE_INFO;
    WRITE (FILE => BANK_FILE,
           ITEM => CHANGE_BANK_REC,
           FROM => CHANGE_POS);
  end if;
end if;
end BK_CHANGE;
or terminate;
end select;
exit when NO_FILES_PROCESSED = 3;
-- Когда количество закрытых файлов станет
-- равно трем, то входные данные должны
-- закончиться.
end loop;
CLOSE (BANK_FILE);
-- Цикл будет выполняться до тех пор, пока
-- будут открыты три файла, обрабатывающи-
-- еся тремя независимыми задачами.
-- После закрытия этих файлов ни одна задача
-- не будет обращаться к файлу BANK_FILE.
and LOOKING_FOR_BANKS;

```

Тело задачи из данного примера реализует механизм, гарантирующий взаимное исключение некоторых событий. Термин «взаимное исключение» относится к операциям, которые нельзя выполнять одновременно с точки зрения их совместимости. Например, нельзя параллельно обновлять и считывать данные, поскольку одна задача может считать данные, которые уже стали устаревшими из-за того, что другая задача записала новые данные или изменила имеющиеся. Оператор отбора, расположенный в теле задачи, позволяет добиться нужного взаимного исключения, так как в любой момент времени может быть отображена только одна альтернатива, а данные можно считывать или изменять только посредством (составного) оператора отбора select.

**Пример.** Описываемая здесь задача `CHANGE_FILE_1` выполняет обновление содержимого файла `BANK_FILE` на основании информации, считанной из файла `CHNG_FILE`. Эта задача является непосредственной модификацией задач `CHECK_FILE_1` и `CHECK_FILE_2`. Данная задача размещается в той же самой программе, где находились эти две задачи-предшественницы.

```

task CHANGE_FILE_1;

task body CHANGE_FILE_1 is
  CHNG_BNK_1 : STRING ( 1 .. 20 );
  CHNG_INFO_1 : STRING (1 .. 49);
  NO_HITS_FILE_CHNG, TOT_FILE_CHNG : NATURAL := 0;
  BNK_CHNG_POS : NATURAL;
  IS_CHNG_THERE : BOOLEAN;

```

```

CHNG_FILE : FILE_TYPE;
begin
  OPEN(FILE => CHNG_FILE,
        MODE => IN_FILE,
        NAME => "CHANG1.DAT",
        FORM => "");
  while not END_OF_FILE (CHNG_FILE)
  loop
    READ (FILE => CHNG_FILE,
          ITEM => CHNG_BNK_1 );
    TOT_FILE_CHNG := TOT_FILE_CHNG + 1;
    LOOKING_FOR_BANKS.BK_CHANGE
      (CHANGE_BANK_NAME => CHNG_BNK_1,
       CHANGE_POS       => BNK_CHNG_POS,
       CHANGE_INFO      => CHNG_INFO_1,
       CHANGE_FOUND     => IS_CHNG_THERE );
    if IS_CHNG_THERE
    then
      NO_HITS_FILE_CHNG := NO_HITS_FILE_CHNG + 1;
    end if;
  end loop;
  LOOKING_FOR_BANKS.BK_CHANGE
    (CHANGE_BANK_NAME => "12345678901234567890",
     CHANGE_POS       => BNK_CHNG_POS,
     CHANGE_INFO      => CHNG_INFO_1,
     CHANGE_FOUND     => IS_CHNG_THERE );
  PUT(" The number of banks in CHNG_FILE is ");
  PUT( TOT_FILE_CHNG );
  PUT(" The number of banks in CHNG_FILE " &
       "and BANK_FILE is ");
  PUT(NO_HITS_FILE_CHNG);
  CLOSE (CHNG_FILE);
end CHANGE_FILE_1;

```

### 10.2.3. Оператор отбора select: условные и таймированные вызовы входов

Операторы селективного ожидания определяют условия, при которых станет возможным rendezv с вызываемой задачей, а операторы условного и таймированного вызовов входа можно использовать в вызывающих задачах для попытки установления rendezv, которое может быть при некоторых обстоятельствах отменено. При *условном вызове входа* попытка установления rendezv отменяется немедленно, если оно оказывается невозможным. При *таймированном вызове входа* обращение к нему отменяется после заданной задержки.

Полная форма условного вызова входа:

```

select
  оператор_вызова_входа
  последовательность_операторов
else
  последовательность_операторов
end select;

```

Член «последовательность\_операторов», расположенный после оператора\_вызова\_входа, может отсутствовать.

Оператор условного вызова входа выполняется следующим образом. Вычисляются фактические параметры оператора вызова входа (если они есть). Затем предпринимается попытка установить randevu с вызываемой задачей. Если немедленное осуществление randevu невозможно, то вне зависимости от причины этого будет выполняться else-часть данного оператора. Если же randevu произойдет, то после его окончания будет выполняться последовательность операторов, стоящая за оператором вызова входа (если она присутствует).

Условный вызов входа дает возможность сэкономить некоторое время и выполнить какие-то полезные действия, если немедленное установление randevu окажется невозможным<sup>1)</sup>. Вызов не помещается в очередь. Вместо этого будет выполняться последовательность операторов, расположенная после зарезервированного слова else, а попытку вызова входа можно будет повторить позднее. В упр. 2 в конце данной главы предлагается модифицировать текст программы GATHER\_BANK\_STATISTICS, с тем чтобы использовать условные вызовы входов.

Полная форма таймированного вызова входа:

```
select
оператор_вызова_входа
последовательность_операторов
or
альтернатива_с_задержкой
последовательность_операторов
end select;
```

Как и в случае условного вызова входа, здесь последовательность операторов, располагающаяся после оператора вызова входа, может отсутствовать. Может не быть и альтернативы с задержкой.

Выполнение оператора таймированного вызова входа происходит следующим образом. Вычисляются фактические параметры оператора вызова входа (если они есть). Затем вычисляется выражение, стоящее после зарезервированного слова delay. После этого в течение вычисленного интервала времени предпринимается попытка установления randevu с вызываемой задачей. Если randevu станет возможным, то оно начнется. После завершения randevu выполняется последовательность операторов (если она имеется), стоящая после оператора вызова входа. Если же в течение заданного периода времени установить randevu окажется невозможным, то выполнится последовательность операторов (если она есть), стоящая после оператора задержки.

Если при условном или таймированном вызове входа обнаружится, что вызываемая задача уже завершилась, то возникнет исключительная ситуация TASKING\_ERROR. Пример того, как можно действовать в такой ситуации, будет дан в гл. 11.

### 10.2.4. Атрибуты входа и задачи

В нижеследующей программе приводится пример употребления таймированных вызовов входов. В программе моделируется в упрощенном виде производственная задача, в которой изделие при изготовлении проходит две производственные операции — OPER1 и OPER2. В программе также используется запрос атрибута входа E'COUNT, который дает количество обращений ко входу E задачи T, стоящих в данный момент в очереди.

Для типов «задача» и объектов этих типов существуют и некоторые другие полезные атрибуты (T обозначает объект типа «задача»):

<sup>1)</sup> Не тратя времени на ожидание. — *Прим. перев.*

T'CALLABLE	Дает логическое значение FALSE, если задача закончилась (полностью завершилась или находится на стадии аварийного завершения). В остальных случаях получается значение TRUE
T'TERMINATED	Дает логическое значение TRUE, если задача T полностью завершилась. В противном случае вырабатывается значение FALSE

На производственном участке имеются три машины: MACH1, MACH2 и MACH3, которые выполняют производственные операции. Машина MACH1 выполняет операцию OPER1 приблизительно за 3 мин, MACH2—операцию OPER2 примерно за 7 мин, а MACH3—обе операции, т.е. OPER1 и OPER2 (всегда последовательно), приблизительно за 11 мин. После окончания изготовления изделие подлежит проверке, которая занимает около 2 мин. При этом отбраковываются 5% изделий. Каждые 2 ч контролер уходит на пятнадцатиминутный перерыв выпить чашечку кофе. (Но он уходит на перерыв только после окончания проверки очередного изделия, а не в середине процесса проверки.) Забракованные изделия должны повторно пройти производственные операции OPER1 и OPER2.

Указанные здесь затраты времени на производственные операции и проверку являются усредненными. Реальные затраты времени распределены по некоторому предполагаемому случайному закону. Для получения фактических затрат времени используется пакет DISTRIBUTIONS.

Предположим, что в начале рабочего дня поступила партия из двухсот заготовок изделий. Программа должна определить, сколько времени будет потрачено на производство готовых изделий из этой партии заготовок.

#### Программа PLANT\_SCHED

```
with CALENDAR; use CALENDAR;
package DISTRIBUTIONS is
  type DIST_KINDS is (TIME_MACH1, TIME_MACH2,
                     TIME_MACH3, INSPECTION);
  -- Для сокращения затрат времени, затраты
  -- взятого на моделирование, случайные вели-
  -- чины измеряются в секундах, а не в
  -- минутах.
  function PROBABILITY(FORM_DIST : in out
    DIST_KIND) return DURATION;
  function INSP_RESULT return BOOLEAN;
  -- Эта функция вырабатывает значение TRUE,
  -- если изделие успешно проходит проверку.
end DISTRIBUTIONS;

--
with DISTRIBUTIONS; use DISTRIBUTIONS;
with CALENDAR; use CALENDAR;
-- Этот пакет нужен для функций CLOCK и "-".
with TEXT_IO; use TEXT_IO;
procedure PLANT_SCHED is
  -- В упр.4 в конце главы предлагается иной
  -- подход - все двести изделий моделируются
  -- семейством задач.
  N_OF_FINISHED_PROD : NATURAL := 0;
  N_OF_REJECTS : NATURAL := 0;
  MACH1_AVAIL, MACH2_AVAIL, MACH3_AVAIL :
    NATURAL := 0;
  -- С помощью этих переменных подсчитывается ко-
  -- личество изделий, обработанных каждой машиной,
  -- но не принятых немедленно следующей стадией
```



```

-- производственного процесса. Это - разделяемые
-- переменные, их "поведение" будет подробнее
-- рассмотрено в следующем разделе.
task COORDINATOR is
  entry SCHED;
end COORDINATOR;
task MACH1 is
  entry OPER1;
end MACH1;
task MACH2 is
  entry OPER2;
end MACH2;
task MACH3 is
  entry OPER1_N_2;
end MACH3;
task STOP_INSPECTION;
task INSPECT is
  entry PRODUCT;
  entry TAKE_A_BREAK;
end INSPECT;
--
task body MACH1 is
  MACH1_JOB_DURATION : DURATION;
begin
  MACH1_JOB_DURATION := PROBABILITY(TIME_MACH1);
loop
  accept OPER1;
  delay MACH1_JOB_DURATION;
  -- Оператор отбора, расположенный ниже, га-
  -- рантирует то, что эта задача не будет
  -- приостановлена (и соответственно данная
  -- машина не будет простаивать) в случае, если
  -- вторая машина окажется недоступной. Анало-
  -- гично обстоит дело со всеми тремя машинами.
  select
    MACH2.OPER2;
  else
    MACH1_AVAIL := MACH1_AVAIL + 1;
  end select;
  MACH1_JOB_DURATION := PROBABILITY(TIME_MACH1);
end loop;
end MACH1;
--
task body MACH2 is
  MACH2_JOB_DURATION : DURATION;
begin
  MACH2_JOB_DURATION := PROBABILITY(TIME_MACH2);
loop
  loop
    -- Данный оператор отбора обеспечивает
    -- отсутствие простоя машины MACH2, если
    -- нет вызова от MACH1. В упр.3 в конце
    -- главы предлагается реализовать иной
    -- подход, предусматривающий использо-
    -- вание задачи-буфера.
    select
      accept OPER2;
    exit;
  end select;
end loop;
end MACH2;

```

```

    else
      if MACH1_AVAIL > 0
        then
          MACH1_AVAIL := MACH1_AVAIL - 1;
          exit;
        end if;
      end select;
    end loop;
    delay MACH2_JOB_DURATION;
    select
      INSPECT.PRODUCT;
    else
      MACH2_AVAIL := MACH2_AVAIL + 1;
    end select;
    MACH2_JOB_DURATION := PROBABILITY(TIME_MACH2);
  end loop;
end MACH2;
--
task body MACH3 is
  MACH3_JOB_DURATION : DURATION;
begin
  MACH3_JOB_DURATION := PROBABILITY(TIME_MACH3);
  loop
    accept OPER1_N_2;
    delay MACH3_JOB_DURATION;
    select
      INSPECT.PRODUCT;
    else
      MACH3_AVAIL := MACH3_AVAIL + 1;
    end select;
    MACH3_JOB_DURATION := PROBABILITY(TIME_MACH3);
  end loop;
end MACH3;
--
task body COORDINATOR is
begin
  -- До тех пор пока будут иметься вызовы к данной
  -- задаче или до тех пор пока количество забра-
  -- кованных изделий будет положительным, таймиро-
  -- ванный вызов входа будет предпринимать попытку
  -- установления rendezv с задачей MACH1 или MACH3
  -- путем переключения от одной задачи к другой,
  -- если rendezv окажется невозможным, каждые 0.01
  -- с. Здесь моделирование вносит некоторое
  -- добавочное время ввиду возможной задержки на
  -- 0.01 с.
  accept SCHED;
  -- Данный оператор приема гарантирует то, что обра-
  -- ботка вызовов начнется с главной программы.
  loop
    loop
      select
        MACH1.OPER1;
      exit;
    or
      delay 0.01;
    end select;
  select

```

```

MACH3.OPER1_N_2;
exit;
or
  delay 0.01;
end select;
-- Альтернативный способ моделирования данной
-- ситуации (без дополнительной задержки на
-- 0.01с) - это использование условного выхо-
-- за входа:
-- loop
--   select
--     MACH1.OPER1;
--     exit;
--   else
--     null;
--   end select;
--   select
--     MACH3.OPER1_N_2;
--     exit;
--   else
--     null;
--   end select;
-- end loop;
end loop;
select;
  accept SCHED;
else
  -- Эта альтернатива выбирается в том случае,
  -- когда все 200 изделий обработаны, а не-
  -- сколько бракованных изделий необходимо
  -- обработать заново. Здесь можно было бы
  -- просмотреть количество бракованных из-
  -- делий с целью принятия решения о направ-
  -- лении их на повторную обработку. Эта
  -- возможность обсуждается в следующем
  -- разделе.
  null;
end select;
end loop;
end COORDINATOR;
--
task body STOP_INSPECTION is
begin
  loop
    delay 120.00;
    INSPECT.TAKE_A_BREAK;
  end loop;
end STOP_INSPECTION;
--
task body INSPECT is
  INSP_PASSED : BOOLEAN;
  EXAM_TIME : DURATION;
  START_MANUFACT, STOP_MANUFACT : TIME;
  DURATION_MANUFACT : DURATION;
  package DURATION_IO is new FIXED_IO
    (DURATION);
  use DURATION_IO;
begin
  -- Как упоминалось ранее, DURATION - это

```

```

-- predetermined part of predetermined
-- environment. CLOCK - this is one of the func-
-- tion of the CALENDAR package, it gives the value
-- of the current time (type TIME).
START_MANUFACT := CLOCK;
INSP_PASSED := INSP_RESULT;
EXAM_TIME := PROBABILITY(INSPECTION);
loop
  -- This cycle is intended to ensure the guarantee
  -- that the check will start only in
  -- the case, if there are products, subject-
  -- ed to the check.
  -- Here you can use the task-buffer
  -- (see. ex. 3 at the end of the chapter).
  loop
    select
      accept PRODUCT;
    exit;
  else
    if MACH2_AVAIL > 0
    then
      MACH2_AVAIL := MACH2_AVAIL - 1;
      exit;
    end if;
    if MACH3_AVAIL > 0
    then
      MACH3_AVAIL := MACH3_AVAIL - 1;
      exit;
    end if;
  end select;
end loop;
delay EXAM_TIME;
if INSP_PASSED
then
  N_OF_FINISHED_PROD := N_OF_FINISHED_PROD
    + 1;
else
  N_OF_REJECTS := N_OF_REJECTS + 1;
  -- With these two variables you have access
  -- to some other tasks. For example,
  -- the task COORDINATOR can check the
  -- values for the purpose of estab-
  -- lishing, whether it is necessary to stop the plan-
  -- ning of the processing of the products.
end if;
EXAM_TIME := PROBABILITY(INSPECTION);
INSP_PASSED := INSP_RESULT;
exit when N_OF_FINISHED_PROD := 200;
-- Should the controller make a break?
if TAKE_A_BREAK'COUNT > 0
-- This condition will be true, if
-- there is an expected call. This means
-- that it is time to make a break.
then
  accept TAKE_A_BREAK;
  delay 15;
end if;
-- The operator if can be replaced by

```

```

-- оператор отбора, например:
-- select
--   accept TAKE_A_BREAK;
--   delay 15;
-- else;
--   null;
-- end select;
end loop;
-- Здесь можно было бы использовать несколько
-- операторов прекращения задачи (см. следую-
-- щий раздел) для завершения приостановленных
-- задач, ожидающих вызова.
STOP_MANUFACT := CLOCK;
DURATION_MANUFACT := START_MANUFACT -
                        STOP_MANUFACT;
-- Обратите внимание, что операция "-" пере-
-- крыта, а из контекста видно, что она зада-
-- ется функцией, определенной в пакете CALENDAR.
PUT (DURATION_MANUFACT, 10, 2);
end INSPECT;
begin
  for I in 1 .. 200
    loop
      COORDINATOR.SCHED;
    end loop;
    -- Предпринять попытку планирования изготав-
    -- ления двухсот изделий. Фактически понадо-
    -- бится изготовить больше изделий, так как
    -- некоторые из них будут забракованы.
  end PLANT_SCHED;

```

Работу различных задач программы PLANT\_SCHED иллюстрирует рис. 10.3. На шаге 1 задача-координатор COORDINATOR только что обнаружила, что задача MACH1 свободна, в то время как задача MACH2 ожидает поступления заготовки изделия, а задача MACH3 пытается установить randevу с задачей INSPECTOR. На шаге 3 все задачи работают независимо друг от друга.

### 10.2.5. Операторы прекращения задачи abort

В Аде есть средство для явного завершения задач — *оператор прекращения задачи abort*. Этот оператор состоит из зарезервированного слова abort; за ним следуют список имен задач, выполнение которых нужно прекратить, и точка с запятой. Имена задач разделены запятыми. Например:

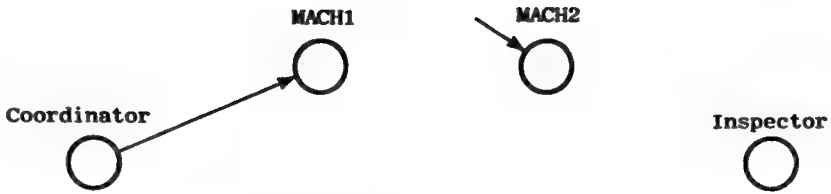
```
abort TASK1, TASK2;
```

Здесь должно быть прекращено выполнение задач TASK1 и TASK2. Вначале любая прекращаемая задача попадает в состояние *аварийного завершения*, которое является состоянием, предшествующим полному завершению. Задача, находящаяся в этом состоянии, больше не может участвовать в новых randevу. Любая задача, зависящая от задачи, находящейся в состоянии аварийного завершения, сама попадает в это состояние, если только она уже не завершилась полностью. Однако задаче, находящейся в состоянии аварийного завершения, разрешено закончить randevу, которое началось до попадания ее в это состояние.

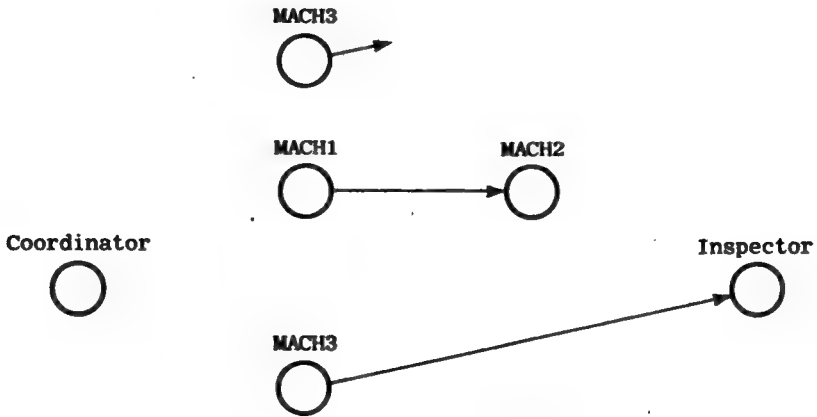
Если задача попадет в состояние аварийного завершения в то время, когда она приостановлена или находится в ожидании по оператору задержки, то она закончится немедленно. В противном случае эта задача закончится в момент достижения ею

Время

Шаг 1



Шаг 2



Шаг 3

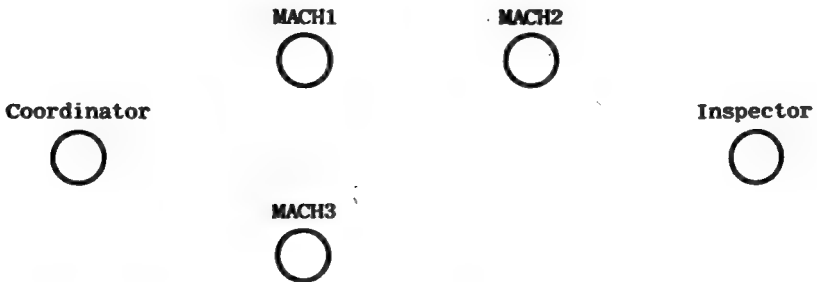


Рис. 10.3. Возможные рандеву в программе PLANT\_SCHED.

некоторой точки синхронизации, такой, как зарезервированное слово end, другой оператор abort, начало или конец оператора ассерт, оператор select или обработчик исключительной ситуации (см. гл. 11).

Задача может прекращать любую другую задачу, имя которой ей известно в соответствии с правилами видимости, включая саму себя. Это весьма мощное средство, использование которого программист должен тщательно обосновывать.

## 10.3. РАЗДЕЛЯЕМЫЕ ПЕРЕМЕННЫЕ И ИНСТРУКЦИЯ ТРАНСЛЯТОРУ PRIORITY

### 10.3.1. Разделяемые переменные

Вообще говоря, не следует иметь две или более задачи, которые считывают и изменяют значение одной и той же переменной. Переменная, которая доступна для нескольких задач и значение которой может быть изменено этими задачами, называ-

ется *разделяемой переменной*<sup>1)</sup>. Она может принять новые значения, в то время как некоторые из задач будут оперировать с другими (более старыми) ее значениями. Такая ситуация может возникнуть из-за того, что задачи независимы друг от друга и, как считается, выполняются параллельно. Единственный случай, когда две задачи могут получить доступ к одной и той же копии значения переменной, — это когда они синхронизированы.

Однако в каждой реализации языка Ада должны быть предусмотрены некоторые дополнительные правила, применяемые только к разделяемым переменным скалярных или ссылочных типов. Эти правила делают несколько менее непредсказуемым употребление разделяемых переменных скалярного или ссылочного типа, так как они обеспечивают в конкретной реализации Ады соблюдение следующего условия. Если задача Т считывает разделяемую переменную в интервале между двумя соседними точками синхронизации, скажем, А и В, то никакой другой задаче не разрешается изменять значение этой переменной в период, когда задача Т прошла точку А и еще не дошла до точки В. Аналогично, если задача Т изменяет значение разделяемой переменной на интервале между А и В, никакой другой задаче не разрешается ни считывать, ни изменять значение этой разделяемой переменной в период, когда задача Т прошла точку А, но еще не дошла до В.

Например, переменные `N_OF_FINISHED_PROD` и `N_OF_REJECTS` в программе `PLANT_SCHED` являются разделяемыми переменными скалярного типа. Одну из них изменяет задача `INSPECT`. В соответствии с правилами, регулирующими использование разделяемых переменных, никакая другая задача не сможет ни изменять, ни считывать значение переменной `N_OF_FINISHED_PROD` или `N_OF_REJECTS` в период, когда в задаче `INSPECT` будут выполняться операторы, расположенные между точкой синхронизации `accept PRODUCT` и следующей точкой синхронизации. Следующей точкой синхронизации может быть та же самая точка `accept PRODUCT`, новая точка, например `accept TAKE_A_BREAK` или окончание задачи. В данном случае правила использования разделяемых переменных гарантируют правильность проверки в задаче `COORDINATOR` на нулевое количество забракованных изделий. Задаче `COORDINATOR` будет доступна только обновленная копия значения переменной `N_OF_REJECTS`, что предотвращает планирование изготовления большего числа изделий, чем требуется.

Обратите внимание, что в последней версии задачи `LOOKING_FOR_BANKS` (см. разд. 10.2.) переменные `CHANGE_HEAD` и `CHANGE_BANK_REC` не являются разделяемыми, так как только задача `LOOKING_FOR_BANKS` может обращаться к ним. Поэтому другие задачи не могут помешать обновлению элементов файла `BANK_FILE`. Если же эти переменные комбинированного типа сделать доступными для других задач (например, объявив их перед всеми спецификациями задач программы `GATHER_BANK_STATISTICS`), то правила, регулирующие использование скалярных и ссылочных разделяемых переменных, не будут действовать. В этом случае изменение упомянутых структур приведет к еще более непредсказуемым последствиям.

В ряде случаев, однако, употребление разделяемых переменных представляется оправданным. Рассмотрим, к примеру, программу `POSTING_PROC` из гл. 8. В этой программе можно использовать несколько задач — быть может, по одной задаче на каждый день, когда совершались сделки. Поэтому можно будет перейти к параллельной регистрации сделок. Регистрация начинается параллельным считыванием из файла `TRANS_FILE` сведений о нескольких днях, а заканчивается записью соответствующих номеров сделок в элементы файла, ассоциируемые с банками, участвующими в сделке.

Опять-таки можно представить, что при регистрации сделок будет работать

<sup>1)</sup> Ее также называют общей нелокальной переменной. — *Прим. перев.*

несколько задач, обслуживающих одновременно несколько банков. Можно использовать переменную, которая потребуется каждой задаче, выполняющей регистрацию в файле BANK\_FILE. Эта переменная должна содержать значение, отражающее последнее размещение элементов файла BANK\_FILE, с тем, чтобы правильно записывать данные о сделках для конкретных банков. Было бы, однако, ошибкой использовать только функцию SIZE, так как к тому моменту, когда задача будет готова записать новый элемент в файл BANK\_FILE, размер файла уже может отличаться от первоначально используемого размера. В качестве альтернативного способа решения этой проблемы может служить введение другой задачи, целиком ответственной за размещение новых элементов в файле BANK\_FILE. Это иллюстрируется следующим примером.

**Пример.** Для данного примера выбрана задача POST\_INDIV\_BANK из программы POSTING\_PROC (см. гл. 8). Она переписана как тип «задача». Вначале представим текст задачи OBTAIN.

```
task OBTAIN is
  -- Эта задача выдает позицию в файле для
  -- нового элемента.
  entry FILE_ELEM(FORM_TR_POS : out POSITIVE_COUNT);
end OBTAIN;
--
task body OBTAIN is
  DUMMY_BANK_REC : BANK_REC(POSTING_DATA) :=
    ( POSTING_DATA, (1 .. 12) => (0,0,1), 0);
  -- Эта структура нужна для записи в файл пустого
  -- элемента.
begin
  accept FILE_ELEM (FORM_TR_POS : out POSITIVE_COUNT)
  do
    if SIZE(BANK_FILE) <= GLOBAL_NO_OF_BANKS
    then
      FORM_TR_POS := GLOBAL_NO_OF_BANKS + 1;
    else
      FORM_TR_POS := SIZE(BANK_FILE) + 1;
    end if;
    WRITE(BANK_FILE, DUMMY_BANK_REC, FORM_TR_POS);
    -- Запись пустого элемента.
  end;
end OBTAIN;
```

Теперь преобразуем задачу POST\_INDIV\_BANK из программы POSTING\_PROC в тип «задача» с именем PARALLEL\_POST\_INDIV\_BANK. При этом используем задачу OBTAIN.

```
task type PARALLEL_POST_INDIV_BANK is
  entry TRANSACT (FORM_TRANS : TRANS_NUMBER;
                  FORM_BANK : STRING(1..20));
end PARALLEL_POST_INDIV_BANK;
--
task body PARALLEL_POST_INDIV_BANK is
  BK_POS, TR_POS, SAVE_TR_POS : POSITIVE_COUNT;
  BK_FOUND : BOOLEAN;
  ANY_BANK : STRING(1..20);
  ANY_TRANS : TRANS_NUMBER;
  ANY_BANK_REC : BANK_REC;
  ANY_POST_REC : BANK_REC;
```



```

WRK_POST_INFO : POST_INFO;
begin
  accept TRANSACT (FORM_TRANS : TRANS_NUMBER;
                  FORM_BANK : STRING )
  do
    -- Копирование значений формальных параметров,
    -- которые будут использоваться в задаче.
    ANY_BANK := FORM_BANK;
    ANY_TRANS := FORM_TRANS;
  end;
  RETRIEVE (ANY_BANK, BK_POS, BK_FOUND);
  if BK_FOUND
  then
    READ (BANK_FILE, ANY_BANK_REC, BK_POS);
    if ANY_BANK_REC.FIRST_TRAN = 0
    then
      -- Это условие будет истинно, если для данного
      -- банка не зарегистрировано ни одной сделки.
      OBTAIN.FILE_ELEM (TR_POS);
      -- Этот оператор заменяет оператор "if", уста-
      -- навливающий положение последнего элемента
      -- файла.
      ANY_BANK_REC.FIRST_TRAN := TR_POS;
      ANY_BANK_REC.LAST_TRAN := TR_POS;
      ANY_BANK_REC.LAST_POS := 1;
      WRK_POST_INFO := (ANY_TRANS,
                        2 .. 12 => (1,1,1));
      ANY_POST_REC := (POSTING_DATA, WRK_POST_INFO, 0);
    else
      if ANY_BANK_REC.LAST_POS = 12
      then
        -- Здесь задается целиком строка регист-
        -- рационной информации. Оператор
        -- TR_POS := SIZE(BANK_FILE) + 1;
        -- заменяется на следующие операторы:
        OBTAIN.FILE_ELEM(TR_POS);
        READ(BANK_FILE, ANY_POST_REC,
            ANY_BANK_REC.LAST_TRAN);
        ANY_POST_REC.NXT_PST_LINE := TR_POS;
        WRITE(BANK_FILE, ANY_POST_REC,
            ANY_BANK_REC.LAST_TRAN);
        ANY_BANK_REC.LAST_TRAN := TR_POS;
        ANY_BANK_REC.LAST_POS := 1;
        WRK_POST_INFO := (ANY_TRANS,
                          2 .. 12 => (1,1,1));

        ANY_POST_REC := ( POSTING_DATA,
                          WRK_POST_INFO, 0 );
      else
        -- Здесь есть место.
        TR_POS := ANY_BANK_REC.LAST_TRAN;
        READ ( BANK_FILE, ANY_POST_REC,
            ANY_BANK_REC.LAST_TRAN);
        ANY_BANK_REC.LAST_POS :=
            ANY_BANK_REC.LAST_POS + 1;
        ANY_POST_REC.POSTING_LINE (
            ANY_BANK_REC.LAST_POS ) := ANY_TRANS;
      end if;
    end if;
  end if;

```

```

end if;
WRITE ( BANK_FILE, ANY_BANK_REC, BK_POS );
WRITE ( BANK_FILE, ANY_POST_REC, TR_POS );
else
  PUT(" Bank not found ");
  PUT( ANY_BANK );
end if;
end PARALLEL_POST_INDIV_BANK;

```

### 10.3.2. Программа, в которой употребляются разделяемые переменные

В следующей программе показано, как можно практически использовать типы «задача» и ссылочные типы, указывающие на объекты «задача». Эта программа представляет собой модификацию программы POSTING\_PROC, при которой пользователю разрешается генерировать столько регистрирующих задач, сколько имеется дней, в которые заключались сделки.

#### Программа SEVERAL\_POSTING\_PROC

```

with TEXT_IO; use TEXT_IO;
with TRANSACTION_RESOURCES;
use TRANSACTION_RESOURCES;
with BANK_RESOURCES; use BANK_RESOURCES;
-- Предполагается, что эти пакеты, определенные
-- в гл.8, оттранслированы и доступны данной
-- программе.
procedure SEVERAL_POSTING_PROC is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  CURR_TRANS_REC_1, CURR_TRANS_REC_OTHER :
    TRANS_REC;
  YY_AND_MM : STRING(1 .. 4);
  MM_ONLY : STRING(1 .. 2);
  EXT_NAME : STRING(1 .. 8);
  WORK_POSITIVE : NATURAL;
  CURR_TRANS_HEADER : TRANS_HEADER;
  CURR_TRANS_INDEX : POSITIVE_INDEX;
  NO_OF_DAILY_JOBS : NATURAL;
  -- До этой точки прежние об'явления останутся
  -- без изменений.
  procedure POST_BNK_PAR(FORM_TRANS_REC : TRANS_REC;
    FORM_DAY : NATURAL) is
    LOCAL_FULL_TRANS_NO : TRANS_NUMBER;
  begin
    -- Сюда следует поместить последовательность
    -- операторов процедуры POST_BNK из программы
    -- POSTING_PROC.
  end POST_BNK_PAR;
  --
  task type DAY_POSTING is
    -- Этот тип "задача" будет использоваться при
    -- генерировании задач-об'ектов, по одной на
    -- каждый день. Задачи будут запускаться для
    -- считывания данных о сделках дня.
    entry GET_THIS_DAY(I : INTEGER range 1..31);
  end DAY_POSTING;
  --

```

```

type DAY_POST_PTR is access DAY_POSTING;
-- Этот ссылочный тип указывает на тип задач
-- DAY_POSTING.
type MONTHLY_COLLECTION_OF_JOBS is array
  (1 .. 31) of DAY_POST_PTR;
-- Это - регулярный тип, элементы которого служат
-- указателями на задачи.
ACT_MONTHLY_JOB : MONTHLY_COLLECTION_OF_JOBS;
--
task body DAY_POSTING is
  --
  -- Данное тело задачи представляет собой модифи-
  -- кацию наиболее глубоко вложенного цикла
  -- исполняемой части процедуры POSTING_PROC.
  -- Вначале обозначение дня запоминается с помощью
  -- переменной THIS_DAY. Затем считываются все
  -- сделки, соответствующие дню THIS_DAY, а соот-
  -- ветствующая информация передается для обра-
  -- ботки в процедуру POST_BNK_PAR. Здесь более
  -- удобным было бы применить задачу-буфер, что,
  -- возможно, позволило бы отказаться от создания
  -- большой очереди приостановленных задач..
  THIS_DAY_TRANS_HEADER : TRANS_HEADER;
  THIS_DAY_TRANS_INDEX : POSITIVE_INDEX;
  THIS_DAY : INTEGER range 1 .. 31;
  DAILY_TRANS_REC : TRANS_REC;
begin
  accept GET_THIS_DAY(I : INTEGER range 1 .. 31)
  do
    -- Зарегистрируем сделки этого дня.
    THIS_DAY := I;
  end GET_THIS_DAY;
  THIS_DAY_TRANS_HEADER :=
    CURR_TRANS_REC.1.HEADER_LINE.TRANS_STATUS
    (THIS_DAY);
  THIS_DAY_TRANS_INDEX :=
    THIS_DAY_TRANS_HEADER.FIRST_TRANS;
  if THIS_DAY_TRANS_INDEX /= 1
  then
    READ(TRANS_FILE, DAILY_TRANS_REC,
      THIS_DAY_TRANS_INDEX);
    -- Прочитана информация о первой сделке дня.
    while DAILY_TRANS_REC.REG_LINE.TRAN_NEXT /= 0
    loop
      -- Здесь вызывается процедура POST_BNK_PAR.
      POST_BNK_PAR (DAILY_TRANS_REC, THIS_DAY);
      THIS_DAY_TRANS_INDEX :=
        DAILY_TRANS_REC.REG_LINE.TRAN_NEXT;
      READ(TRANS_FILE, DAILY_TRANS_REC,
        THIS_DAY_TRANS_INDEX);
    end loop;
    -- Следующий вызов выполняется только для
    -- последней сделки.
    POST_BANK_PAR (DAILY_TRANS_REC, THIS_DAY);
  end if;
  -- Конец задачи.
end DAY_POSTING;
-- Сюда следует поместить спецификацию и тело задачи

```

```

-- PARALLEL_POST_INDIV_BANK, а также спецификации и
-- тела задач, работающих с буфером (см. упр.5 в
-- конце данной главы).
begin
  GET (YY_AND_MM);
  MM_ONLY := YY_AND_MM ( 3 .. 4 );
  if YY_AND_MM < "8601" or YY_AND_MM > "8612"
  then
    PUT (" Bad date ");
  else
    EXT_NAME := "FY" & YY_AND_MM & ".DAT";
    if not TRANS_IO.IS_OPEN(TRANS_FILE)
    then
      TRANS_IO.OPEN(FILE => TRANS_FILE,
                    MODE => INOUT_FILE,
                    NAME => EXT_NAME,
                    FORM => "");
      READ(TRANS_FILE, CURR_TRANS_REC_1, 1);
    end if;
    NO_OF_DAILY_JOBS := 0;
    for I in 1 .. 31
      loop
        CURR_TRANS_HEADER :=
          CURR_TRANS_REC_1.HEADER_LINE.TRANS_STATUS(I);
        CURR_TRANS_INDEX :=
          CURR_TRANS_HEADER.FIRST_TRANS;
        if CURR_TRANS_INDEX /= 1
        then
          -- Результат будет равен TRUE, если для за-
          -- данного дня имеются сделки. В этом слу-
          -- чае запустится соответствующая задача.
          ACT_MONTHLY_JOBS (I) := new DAY_POSTING;
          NO_OF_DAILY_JOBS := NO_OF_DAILY_JOBS + 1;
          -- Здесь не приводится полный текст прог-
          -- раммы, счетчик количества активных задач
          -- можно использовать для того, чтобы уста-
          -- новить момент окончания регистрации
          -- всех сделок.
        end if;
      end loop;
    end if;
    -- Файлы будут закрыты только после окончания ре-
    -- гистрации всех сделок.
    TRANS_IO.CLOSE(TRANS_FILE);
    BANK_IO.CLOSE(BANK_FILE);
  end SEVERAL_POSTING_PROC;

```

### 10.3.3. Инструкция транслятору PRIORITY

Инструкция PRIORITY может применяться для указания степени срочности выполнения какой-либо задачи на Аде. Форма инструкции:

```
pragma PRIORITY (статическое_выражение);
```

Инструкция PRIORITY может появляться в спецификации задачи только один раз. Результатом вычисления статического выражения должно быть целое число, попадающее в диапазон значений подтипа PRIORITY из предопределенного пакета SYSTEM (см. приложение В). Чем выше значение приоритета задачи, тем более неотложным является ее выполнение.

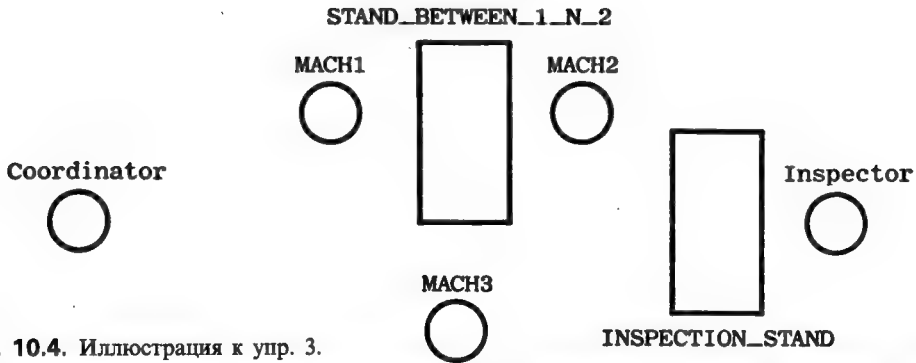


Рис. 10.4. Иллюстрация к упр. 3.

Степень срочности задачи, указываемая в инструкции **PRIORITY**, служит информацией для конкретной системы и помогает ей принять решение о том, какие задачи следует выполнять, если доступные вычислительные ресурсы не позволяют обеспечить одновременное выполнение всех активных задач. Таким образом, ни в какой конкретной реализации не допускается выполнение задачи с низким приоритетом в то время, как задача с высоким приоритетом претендует на выполнение. (Здесь предполагается, что обеим задачам требуются одни и те же ресурсы.)

## УПРАЖНЕНИЯ

1. Модифицируйте программу **GATHER\_BANK\_STATISTICS** таким образом, чтобы она принимала вызовы только после проверки того, что по крайней мере один из файлов **FILE1** и **FILE2** открыт. При этом следует убрать части текста, обрабатывающие значение, служащее признаком конца файла («12345678901234567890»).

2. Перепишите программу **GATHER\_BANK\_STATISTICS** с использованием атрибутов задач и пакета **CALENDAR**. Программа должна выдавать информацию о среднем времени ожидания для вызовов от задач **CHECK\_FILE\_1** и **CHECK\_FILE\_2**.

3. В программе **PLANT\_SCHED** (см. разд. 10.2) в начале членов «последовательность\_операторов» в задачах **MACH1**, **MACH2**, **MACH3** и **INSPECT** применяются циклы. Роль этих циклов — гарантировать то, чтобы каждая задача продолжала независимое выполнение полезных действий, даже если немедленное установление randevu окажется невозможным. Альтернативным и, возможно, лучшим способом реализации такой гарантии было бы наличие задачи-буфера, обеспечивающей связь задач **MACH1** и **MACH2** (назовем ее **STAND\_BETWEEN\_1\_N\_2**) и еще одной задачи-буфера, связывающей **INSPECT** с **MACH2** или **MACH3** (назовем ее **INSPECTION\_STAND**). После завершения действий по **MACH1** первая задача-буфер должна вызвать вход задачи **STAND\_BETWEEN\_1\_N\_2** и зарегистрировать еще одну заготовку изделия, предназначенную для обработки в **MACH2**. (Здесь, по всей видимости, надо употребить обычный счетчик.) В свою очередь задача **MACH2** должна обладать способностью независимо устанавливать randevu с задачей **STAND\_BETWEEN\_1\_N\_2** и решать, стоит ли начинать обработку следующей заготовки изделия, если она имеется. Аналогичные соображения относятся и к задаче **INSPECTION\_STAND**. Взаимосвязи между новыми задачами иллюстрирует рис. 10.4.

4. Предположим, что все 200 заготовок изделий (см. производственную задачу из разд. 10.2) «ожили» и каждая из них стала представлять собой независимую задачу, которая стремится, чтобы ее выполнили. Эти задачи должны вызывать по цепочке либо **MACH1**, **MACH2** и **INSPECT**, либо **MACH3** и **INSPECT**. Разумеется, если изделие будет забраковано, то цепочку вызовов следует повторить. Перепишите программу **PLANT\_SCHED** с использованием этого нового подхода. При этом реализуйте новое условие: ни одно изделие не должно выдавать запрос на обработку, если перед ним уже находятся в ожидании исполнения три вызова. Возможно, здесь следует использовать типы «задача» для обозначения изделий и ссылочные типы, обеспечивающие доступ к объектам типа «задача» для этих изделий.

5. Включите задачу **OBTAIN** и задачу **DAY\_POSTING** (см. разд. 10.3) в текст новой версии программы **SEVERAL\_POSTING\_PROC**. Внесите другие необходимые изменения.

# Исключительные ситуации

## 11.1. ОБЪЯВЛЕНИЕ И ВОЗБУЖДЕНИЕ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

В Аде имеются средства, позволяющие обрабатывать во время выполнения программы ошибки и специально выбранные программные события. Это — механизм исключительных ситуаций. Если происходит ошибка или специально выбранное программное событие, то нормальное последовательное выполнение операторов прерывается и выдается сигнал об ошибке или о событии. Этот процесс называется *возникновением (или возбуждением) исключительной ситуации*. Действия, предпринимаемые при возникновении исключительной ситуации, называются *обработкой исключительной ситуации*. Программист может управлять этими действиями посредством объявления исключительных ситуаций и с помощью специальных участков программы на Аде, называемых обработчиками исключительных ситуаций. Исключительные ситуации могут возникать непредусмотренно из-за непредвиденных ошибок и особых случаев. Но они могут также возбуждаться преднамеренно во время выполнения оператора `raise`. Об этом будет сказано далее в настоящем разделе.

### 11.1.1. Предопределенные исключительные ситуации и исключительные ситуации, определяемые пользователем

Некоторые исключительные ситуации в Аде *предопределены*. Они обычно связаны с ошибками, наиболее часто встречающимися при выполнении программ. Ниже приведен список этих ситуаций:

#### Ошибка

`CONSTRAINT_ERROR`  
(Нарушение\_уточнения)

`NUMERIC_ERROR`  
(Числовая\_ошибка)

`PROGRAM_ERROR`  
(Программная\_ошибка)

#### Условия возникновения исключительной ситуации

- Нарушены диапазон значений, дискриминант или уточнение границ массива, включая попытку использования компоненты структуры, недопустимой для заданного значения дискриминанта
- Предпринимается попытка использовать составное имя, индексированную компоненту, вырезку из массива или атрибут объекта, на который можно указать с помощью ссылочного значения, однако при этом фактическое ссылочное значение — `null`
- При выполнении предопределенной числовой операции не получается правильный результат (точный смысл этой исключительной ситуации будет зависеть от конкретной реализации языка Ада и может включать целое переполнение, деление на ноль или плавающее переполнение)
- При преобразовании значений одного типа к другому произойдет выход за границы диапазона значений
- Выполняется вызов подпрограммы, запуск задачи или обработка родовой конкретизации до того, как обрабо-

таны соответствующие тела

- 1 В процессе вычисления функции достигается ключевое слово end (а не return, как надо бы)
- При выполнении оператора селективного ожидания все альтернативы закрыты, а else-альтернатива отсутствует
- В случаях, обнаружение которых необязательно для Ады, исключительная ситуация определяется конкретной версией языка. Например, программист полагается на какую-то определенную реализацию процессов, о которых в описании Ады говорится, что на такую реализацию надеяться нельзя (в частности, согласование формальных и фактических параметров, которое в разных версиях Ады может быть реализовано с помощью разных механизмов)
- Не хватает памяти для выполнения программы (дополнительная память может выделяться во время выполнения программы при обработке объявлений, при запуске задач и при создании нового объекта при помощи генератора)
- Во время обмена информацией между задачами происходят определенные события (примеры см. разд. 11.3)

#### STORAGE\_ERROR

(Нет\_памяти)

#### TASKING\_ERROR

(Ошибка\_задачи)

Все перечисленные предопределенные исключительные ситуации входят в состав пакета STANDARD (см. приложение В).

Существуют также предопределенные исключительные ситуации ввода-вывода, определенные в пакете IO\_EXCEPTIONS. Они связаны с пакетами ввода-вывода Ады (см. приложение В), и о них будет рассказано в разд. 11.2.

Помимо предопределенных исключительных ситуаций, в Аде есть также *исключительные ситуации, определяемые пользователем*. Эти последние ситуации программист обязан объявлять. Форма объявления исключительных ситуаций такова:

Список\_имен\_исключительных\_ситуаций: exceptions;

Например:

ALARM\_1 : exception;

EMPTY\_FILE, UNEXPECTED\_END\_OF\_FILE : exception;

После записи объявления исключительной ситуации, определяемой пользователем, ее можно возбудить в том случае, когда произойдут некоторые заранее выбранные программные события. Предопределенные исключительные ситуации не нуждаются в объявлениях.

### 11.1.2. Обработчики исключительных ситуаций

Как было упомянуто выше, после возникновения исключительной ситуации управление передается на *обработчик*, предусмотренный для этой ситуации. Наличие или отсутствие такого обработчика определяется намерениями программиста. Если обработчик имеется, то он должен размещаться в конце оператора блока или в конце тела подпрограммы, пакета или задачи.

В программах, приведенных в предыдущих главах, обработчики исключительных ситуаций отсутствовали. Как следствие этого, возбуждение любой предопределенной исключительной ситуации неизбежно должно было бы приводить к прекращению выполнения программы и к распространению этой исключительной ситуации на вызывающую среду. Иными словами, если в процедуре или, скажем, функции возбудится исключительная ситуация, а обработчика для этой ситуации там нет, то исключительная ситуация станет возбуждаться в вызывающей подпрограмме и т. д. до

тех пор, пока не будет достигнута главная программа, или пока не встретится необходимый обработчик исключительной ситуации. Если будет достигнута главная программа, а нужный обработчик не найден, то главная программа завершится и выдаст диагностическое сообщение. Описанный процесс происходил бы в программах из предыдущих глав при возникновении любых предопределенных исключительных ситуаций, поскольку обработчики таких ситуаций в этих программах отсутствуют.

Спецификация обработчиков исключительных ситуаций очень похожа на оператор выбора case и имеет вид:

```
when
имена_исключительных_ситуаций => последовательность_операторов
-- Здесь можно задать еще один или несколько вариантов
-- выбора, аналогичных приведенному выше. Можно указать
-- и альтернативу others («во всех остальных случаях»).
```

Перед самими обработчиками исключительных ситуаций ставится зарезервированное слово exception. После обработчиков помещается слово end, которое отмечает конец оператора блока, конец подпрограммы, задачи или пакета, содержащего обработчик исключительной ситуации.

В качестве примера можно привести такой обработчик исключительных ситуаций (в предположении, что сами исключительные ситуации были объявлены раньше):

```
exception
when NUMERIC_ERROR => PUT(" Bad computation ");
when ALARM_1       => PUT(" Call police ");
when others        => PUT(" Unexpected error ");
end;
```

Как и в случае оператора case, альтернатива others разрешена только для последнего обработчика, и она охватывает все прочие исключительные ситуации (предопределенные и объявленные программистом), не упомянутые в предыдущих обработчиках.

### 11.1.3. Оператор возбуждения исключительной ситуации raise

Оператор raise можно использовать для передачи управления обработчику исключительных ситуаций. Он имеет вид

```
raise имя_исключительной_ситуации;
```

Если оператор raise находится внутри обработчика исключительной ситуации, то при некоторых условиях он может принимать форму:

```
raise;
```

Пример оператора возбуждения:

```
raise ALARM_1;
```

Оператор raise, за которым следует имя\_исключительной\_ситуации, возбудит названную ситуацию. Если имеется соответствующий обработчик, то он получит управление. В противном случае исключительная ситуация будет распространяться по цепочке вызовов.

Вторая форма оператора raise предназначена для повторного возбуждения той же самой исключительной ситуации, которая уже вызвала передачу управления обработчику.



### 11.1.4. Программа, в которой применяются обработчики исключительных ситуаций

Здесь на примере модификации программы DAY\_CONVERSION из разд. 1.4 иллюстрируется обработка исключительных ситуаций. Программа считывает из входного файла несколько строк, в каждой из которых содержится номер дня по Юлианскому календарю и день недели (от MON (понедельник) до SUN (воскресенье)), на который выпало первое января в искомом году. Для каждой входной строки будет выведен день недели, соответствующий указанному номеру дня. Признаком конца данных служит строка с номером дня 0.

#### Программа EXC\_DAY\_CONVERSION

```
with TEXT_IO; use TEXT_IO;
procedure EXC_DAY_CONVERSION is
  type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  package DAY_IO is new ENUMERATION_IO(DAY);
  use DAY_IO;
  type JULIAN is range 1 .. 366;
  JULIAN_DAY : JULIAN;
  CURRENT_DAY, FIRST_JAN_DAY : DAY;
  CURRENT_DAY_POS : INTEGER;
  -- Далее располагаются два новых объявления,
  -- отсутствующие в тексте исходной программы.
  IN_DAY : NATURAL;
  BAD_IN_DAY : exception;
  -- Данная исключительная ситуация будет возникать
  -- в том случае, когда значение переменной IN_DAY
  -- будет превышать 366.
begin
  GET(IN_DAY, 3);
  -- Если значение IN_DAY окажется отрицательным,
  -- то возникнет исключительная ситуация.
  while IN_DAY /= 0
  loop
    if IN_DAY > 366
    then
      -- Пример использования оператора возбуждения
      -- исключительной ситуации raise .
      raise BAD_IN_DAY;
    else
      --** Начало последовательности операторов, иден-
      -- тичных операторам из программы DAY_CONVERSION.
      JULIAN_DAY : IN_DAY;
      -- Если значение величины типа IN_DAY выйдет за
      -- пределы диапазона 0 .. 366, то возникнет
      -- исключительная ситуация CONSTRAINT_ERROR.
      GET(FIRST_JAN_DAY);
      CURRENT_DAY_POS := JULIAN_DAY mod 7 +
        DAY'POS(FIRST_JAN_DAY);
      if CURRENT_DAY_POS > 7
      then
        CURRENT_DAY_POS := CURRENT_DAY_POS - 7;
      end if;
      CURRENT_DAY := DAY'VAL(CURRENT_DAY_POS);
```

```

-- Если значение переменной CURRENT_DAY_POS
-- выйдет за пределы диапазона 1 .. 7, то
-- возникнет исключительная ситуация
-- CONSTRAINT_ERROR (хотя в данном операторе
-- такая ошибка возникнуть не может).
NEW_LINE;
PUT(" This Julian day falls on a ");
PUT(CURRENT_DAY);
if CURRENT_DAY not in Mon .. Fri
then
  PUT(" is a weekend");
else
  PUT(" is a working day ");
end if;
--** Конец последовательности операторов,
-- идентичных операторам программы
-- DAY_CONVERSION.
end if;
SKIP_LINE;
GET(IN_DAY, 3);
end loop;
exception
  when NUMERIC_ERROR | CONSTRAINT_ERROR
    => PUT(" Numeric error or ");
    PUT("Constraint error");
  when BAD_IN_DAY      => PUT(" Bad julian day ");
  when others          =>
    -- Возможно, что во вход-
    -- ных данных есть ошибки.
    PUT(" Other error ");
end EXC_DAY_CONVERSION;

```

### 11.1.5. Возбуждение исключительных ситуаций

Обратите внимание, что одна и та же исключительная ситуация может быть возбуждена в разных местах программного сегмента. Например, в предыдущей программе отмечены по крайней мере две точки, в которых может возникнуть ситуация `CONSTRAINT_ERROR`. Следовательно, в этой программе могут быть сложности с определением того, в каком именно месте возникла исключительная ситуация. Эту трудность можно преодолеть, если поместить критически важные с этой точки зрения операторы в отдельные подпрограммы или операторы блока. Такой вариант действий будет показан в следующем разделе.

Исключительные ситуации могут возбуждаться либо во время выполнения операторов (как в предыдущей программе), либо во время обработки объявлений. Если исключительная ситуация возникает в процессе обработки декларативной части тела пакета, блока, подпрограммы или в процессе обработки объявления пакета, то обработка прекращается и та же самая исключительная ситуация возбуждается повторно, т.е. распространяется. При этом действуют следующие правила:

- если первоначальная исключительная ситуация возникла при обработке декларативной части подпрограммы, то повторная исключительная ситуация возбуждается в точке вызова подпрограммы. Если эта подпрограмма является главной программой на Аде, то выполнение ее заканчивается;
- если первоначально исключительная ситуация возникла в блоке, то затем она возбуждается непосредственно после этого блока;

- если тело или объявление пакета входит в декларативную часть, то исключительная ситуация возбуждается во внешней декларативной части;
- если тело пакета является телом подсегмента, то исключительная ситуация возбуждается в месте расположения соответствующей заглушки;
- если тело или объявление пакета является библиотечным сегментом, то завершается выполнение главной программы.

Если исключительная ситуация возникнет при обработке тела задачи, то обработка прекратится, а задача завершится. В точке запуска задачи возбуждается исключительная ситуация `TASKING_ERROR`. Если же исключительная ситуация возникнет во время обработки объявления задачи, то эта ситуация распространится на внешнюю декларативную часть. Дальнейшие подробности, касающиеся обработки исключительных ситуаций задачами во время выполнения их операторов, будут приведены в разд. 11.3.

### 11.1.6. Подавление проверок

Как говорилось ранее, в Аде есть ряд предопределенных исключительных ситуаций, охватывающих множество возможных событий. К таким ситуациям относится, например, `CONSTRAINT_ERROR`. В некоторых случаях пользователь может пожелать предотвратить возбуждение какой-то конкретной исключительной ситуации для заданных видов событий или для нужных типов в пределах сегмента. Выполнение некоторых проверок, производящихся во время выполнения программ, можно заблокировать при помощи инструкции `SUPPRESS` (Подавить). Форма этой инструкции:

`pragma SUPPRESS (название_проверки);`

Если, вдобавок, программист захочет применить это указание только к конкретному объекту, типу, подпрограмме и т. д., то указание принимает вид

`pragma SUPPRESS (название_проверки, ON => имя);`

Перечислим некоторые из названий проверок для исключительной ситуации `CONSTRAINT_ERROR`:

Название	Применение
<code>ACCESS_CHECK</code> (Проверка_ссылок)	По отношению к составным именам, индексированным компонентам, вырезкам из массивов или к атрибутам объекта, на которые указывает ссылочное значение
<code>DISCRIMINANT_CHECK</code> (Проверка_дискриминанта)	По отношению к значениям дискриминанта; определяет, допустимо ли представленное значение для заданного типа уточнения или существует ли оно для данного объекта комбинированного типа
<code>INDEX_CHECK</code> (Проверка_индекса)	По отношению к границам массивов или к значениям индексов компонент массивов
<code>LENGTH_CHECK</code> (Проверка_длины)	По отношению к компонентам, нуждающимся в согласовании, например при присваивании массивов
<code>RANGE_CHECK</code> (Проверка_диапазона)	По отношению к значениям, для которых требуется проверка их диапазона, к подтипам, которые обрабатываются, к значениям индексов и дискриминантов, требующих проверки соответствия для данного подтипа и т. д.

Приведем некоторые из названий проверок для предопределенной исключительной ситуации `NUMERIC_ERROR`:

Название	Условие выполнения
<code>DIVISION_CHECK</code> (Проверка_деления)	Второй операнд в операциях <code>/</code> , <code>rem</code> или <code>mod</code> равен нулю
<code>OVERFLOW_CHECK</code> (Проверка_переполнения)	Результат числовой операции приводит к переполнению

Инструкция **SUPPRESS**, если она используется, должна находиться в первых строках декларативной части или спецификации пакета. Следует также иметь в виду, что эта инструкция лишь дает разрешение транслятору отменить заданную проверку. Это разрешение может быть не учтено транслятором. Он может не принимать во внимание это указание для тех видов проверок, отмена которых может обойтись слишком дорого.

Пример этой инструкции:

```
pragma SUPPRESS (RANGE_CHECK);
```

Здесь транслятору предоставляется право отменить проверку **RANGE\_CHECK** вообще. Другой пример:

```
pragma SUPPRESS (ACCESS_CHECK, ON => DATE);
```

Здесь транслятору разрешается отменить проверку **ACCESS\_CHECK** для всех объектов типа **DATE**.

## 11.2. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ ВВОДА-ВЫВОДА. ПАКЕТ IO\_EXCEPTIONS

В пакете **IO\_EXCEPTIONS** собраны все исключительные ситуации, определенные для пакетов ввода-вывода **SEQUENTIAL\_IO**, **DIRECT\_IO** и **TEXT\_IO**. Приведем список исключительных ситуаций ввода-вывода из пакета **IO\_EXCEPTIONS** и их краткое описание.

### Исключительная ситуация

### Условия возникновения

**STATUS\_ERROR:** exception;  
(Ошибка\_состояния)

- Предпринимается попытка использовать функции **MODE**, **FORM** и **NAME**, или процедуры **READ**, **WRITE**, **RESET** и **DELETE**, или им подобные при открытом файле
- Делается попытка повторного открытия уже открытого файла

**MODE\_ERROR:** exception;  
(Ошибка\_режима)

- Программа пытается считывать данные из файла с режимом обмена **OUT\_FILE** (выходной)
- Программа пытается определить наличие состояния «конец файла» для выходного файла
- Программа пытается выводить данные в файл с режимом обмена **IN\_FILE** (входной)

#### При использовании пакета **TEXT\_IO**

- Попытка вызова для работы с выходным файлом следующих подпрограмм: **SET\_INPUT**, **SKIP\_LINE**, **END\_OF\_LINE**, **SKIP\_PAGE** и **END\_OF\_PAGE**
- Попытка вызова для работы с входным файлом таких подпрограмм: **SET\_OUTPUT**, **SET\_LINE\_LENGTH**, **SET\_PAGE\_LENGTH**, **LINE\_LENGTH**, **PAGE\_LENGTH**, **NEW\_LINE** и **NEW\_PAGE**

**NAME\_ERROR:** exception  
(Ошибка\_в\_имени)

- Строка, представляющая фактический параметр для процедур **CREATE** или **OPEN**, не позволяет установить связь с внешним файлом. Например, строка может оказаться неприемлемой из-за наличия в ней специальных символов или из-за того, что параметры, указанные в процедуре **OPEN**, не согласуются ни с каким из существующих внешних файлов

**USE\_ERROR:** exception  
(Ошибка\_использования)

- Наступление некоторых событий, несовместимых с характеристиками внешнего файла. Например, программа пытается создать с помощью процедуры **CREATE** новый файл, и параметр, указывающий ре-

DEVICE\_ERROR: exception  
(Сбой устройства)  
END\_ERROR: exception  
(Конец данных)  
DATA\_ERROR: exception;  
(Ошибка в данных)

LAYOUT\_ERROR: exception;  
(Ошибка выходного формата)

жим работы с файлом, задан как IN\_FILE, а файл располагается на устройстве, обеспечивающем только выходные файлы

- В программе используется фактический параметр FORM, несовместимый с параметром FORM, использованным при создании файла
- Обнаружение некоторых видов сбоев аппаратуры во время выполнения операции ввода-вывода<sup>1)</sup>
- Попытка чтения данных после конца файла
- Элемент данных, считываемый процедурой READ, не принадлежит к должному типу. При работе процедуры GET из пакета TEXT\_IO обнаруживается, что входной элемент данных имеет формат, не соответствующий требуемому, или его значение выходит за пределы заданного диапазона

*При использовании пакета TEXT\_IO*

- Вырабатываемые значения параметров COL, LINE или PAGE превышают величину COUNT/LAST
- В строке, являющейся фактическим параметром для PUT, насчитывается слишком много символов
- Предпринимается попытка задать такой номер позиции или такое количество строк в странице, которые превосходят некоторые заранее установленные границы

Пакет IO\_EXCEPTIONS видим в каждом из пакетов ввода-вывода. Все исключительные ситуации, описанные в нем, появляются в обозначениях переименования этих пакетов в таком виде (см. также приложение B):

```
for SEQUENTIAL_IO package :
STATUS_ERROR exception renames
IO_EXCEPTIONS.STATUS_ERROR;
```

Этот формат позволяет проводить различие между исключительными ситуациями для разных пакетов ввода-вывода, так как теперь становится возможным ссылаться на ситуации вида SEQUENTIAL\_IO.NAME\_ERROR или DIRECT\_IO.NAME\_ERROR без использования еще одного дополнительного уточнителя.

Теперь перепишем программу MERGE\_PROC\_GRADES из гл. 8 с тем, чтобы продемонстрировать применение пакета IO\_EXCEPTIONS и показать эффект распространения исключительных ситуаций. В использовании исключительных ситуаций в процедурах READ\_1 и READ\_2 есть некоторая избыточность. В упр. 1 в конце главы предлагается устранить эту избыточность. Существующая же в данной программе избыточность помогает сохранить часть общей структуры программы, разработанной ранее. Это будет способствовать лучшему пониманию новой версии программы.

### Программа EXCEP\_MERGE\_PROC\_GRADES

```
with TEXT_IO; use TEXT_IO;
with SEQUENTIAL_IO;
procedure EXCEP_MERGE_PROC_GRADES is
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
type T_PAIR is
```

<sup>1)</sup> *Примечание.* Данная исключительная ситуация может не применяться в некоторых реализациях Ады: вместо нее сбой оборудования сигнализируются другими ситуациями, например USE\_ERROR.

```

record
  SUBJ_MAT : STRING ( 1 .. 5 );
  SCORE    : NATURAL;
end record;
CURR_PAIR : T_PAIR;
type SEMESTER_TESTS is array (1 .. 25 )
  of T_PAIR;
type BIG_REC is
  record
    BIG_ST_ID : STRING ( 1 .. 10 );
    NO_TESTS  : NATURAL;
    ST_TESTS  : SEMESTER_TESTS;
  end record;
-- Идентификаторы, использованные в тексте,
-- до сих пор совпадают с идентификаторами из
-- программы SEQ_PROC_GRADES и имеют тот же
-- смысл.
package STUD_IO is new SEQUENTIAL_IO(BIG_REC);
use STUD_IO;
CURR_REC_1_IN, CURR_REC_2_IN,
  CURR_REC_OUT : STUD_IO.BIG_REC;
STUD_IN_1, STUD_IN_2, STUD_OUT_FILE :
  STUD_IO.FILE_TYPE;
--
procedure READ_1 is
begin
  READ(FILE => STUD_IN_1,
    ITEM => CURR_REC_1_IN);
  exception
    when STUD_IO.STATUS_ERROR =>
      -- Возможно, не открыт файл.
      PUT(" Status error reading file " &
        "STUDENT-FILE-1-IN.DAT");
      raise;
      -- Повторно возбуждается та же самая
      -- исключительная ситуация
      -- STATUS_ERROR. Она будет распростра-
      -- няться далее на вызывающую подпрог-
      -- рамму. Заметьте, что, если исключите-
      -- льная ситуация об'явлена локально
      -- (например, внутри данной процедуры),
      -- то она не видима извне и, следова-
      -- тельно, на нее нельзя ссылаться
      -- за пределами данной процедуры.
    when STUD_IO.MODE_ERROR =>
      -- Возможно, неправильно
      -- задан режим обмена с файлом,
      -- например данный файл открыт
      -- как OUT_FILE.
      PUT(" Mode error reading file " &
        "STUDENT-FILE-1-IN.DAT");
      raise;
      -- Повторно возбуждается исключительная
      -- ситуация MODE_ERROR. Она будет распро-
      -- страняться на вызывающую подпрограмму.
    when STUD_IO.DEVICE_ERROR =>
      -- Эта ошибка свидетельствует

```

```

-- о сбое аппаратуры.
PUT(" Device error reading file " &
    "STUDENT-FILE-1-IN.DAT");
raise;
-- Здесь невозможна исключительная ситуа-
-- ция NAME_ERROR, поскольку в процедуре
-- READ отсутствует параметр NAME.
-- Ситуация USE_ERROR также не должна
-- здесь возникать, хотя в некоторых
-- реализациях языка она может возбудить-
-- ся при отдельных видах сбоев аппара-
-- туры.
when STUD_IO.END_ERROR =>
    -- Возможно, предпринимается
    -- попытка считывания данных
    -- после конца файла.
    PUT(" End error reading file " &
        "STUDENT-FILE-1-IN.DAT");
    raise;
when STUD_IO.DATA_ERROR =>
    -- Возможно, считанный элемент
    -- не принадлежит к типу
    -- BIG_REC или не распознается
    -- как элемент этого типа.
    PUT(" Data error reading file " &
        "STUDENT-FILE-1-IN.DAT");
    raise;
when others =>
    -- Любые другие исключительные
    -- ситуации.
    PUT(" Unusual kind of error reading " &
        "file " & "STUDENT-FILE-1-IN.DAT");
    raise;
end READ1;
--
procedure READ_2 is
begin
    READ(FILE => STUD_IN_2,
        ITEM => CURR_REC_2_IN);
    exception
    -- Здесь указаны точно те же исключительные
    -- ситуации, что и для процедуры READ1.
    when STUD_IO.STATUS_ERROR =>
        PUT(" Status error reading file " &
            "STUDENT-FILE-2-IN.DAT");
        raise;
    when STUD_IO.MODE_ERROR =>
        PUT(" Mode error reading file " &
            "STUDENT-FILE-2-IN.DAT");
        raise;
    when STUD_IO.DEVICE_ERROR =>
        PUT(" Device error reading file " &
            "STUDENT-FILE-2-IN.DAT");
        raise;
    when STUD_IO.END_ERROR =>
        PUT(" End error reading file " &
            "STUDENT-FILE-2-IN.DAT");
        raise;

```

```

when STUD_IO.DATA_ERROR =>
  PUT(" Data error reading file " &
    "STUDENT-FILE-2-IN.DAT");
  raise;
when others =>
  PUT(" Unusual kind of error reading file " &
    "STUDENT-FILE-2-IN.DAT");
  raise;
end READ2;
--
procedure WRITE_OUT is
begin
  WRITE(FILE => STUD_OUT_FILE,
    ITEM => CURR_REC_OUT );
exception
  when STUD_IO.STATUS_ERROR =>
    -- Возможно, не открыт файл.
    PUT(" Status error writing file " &
      "STUD-MERGED-FILE.DAT");
    raise;
    -- Повторно возбуждается исключитель-
    -- ная ситуация STATUS_ERROR. Она
    -- распространится на вызывающую
    -- подпрограмму.
  when STUD_IO.MODE_ERROR =>
    -- Возможно, неверно задан режим
    -- обмена с файлом как IN_FILE.
    PUT(" Mode error writing file " &
      "STUD-MERGED-FILE.DAT");
    raise;
    -- Повторно возбуждается исключитель-
    -- ная ситуация MODE_ERROR. Она
    -- распространится на вызывающую
    -- подпрограмму.
    -- Исключительная ситуация NAME_ERROR
    -- здесь невозможна, так как процедура
    -- WRITE не имеет параметра NAME.
    -- Исключительная ситуация USE_ERROR
    -- также не должна возбуждаться здесь,
    -- хотя в некоторых реализациях Ады
    -- отдельные виды сбоев аппаратуры
    -- приводят к возбуждению этой исключи-
    -- тельной ситуации.
    -- Ситуация END_ERROR здесь возникнуть
    -- не может.
  when STUD_IO.DATA_ERROR =>
    -- Возможно, записываемый элемент
    -- не относится к типу BIG_REC или
    -- не может быть интерпретирован
    -- как элемент этого типа.
    PUT(" Data error writing file " &
      "STUD-MERGED-FILE.DAT");
    raise;
  when others =>
    -- Остальные исключительные ситуации.
    PUT(" Unusual kind of error writing file " &
      "STUD-MERGED-FILE.DAT");

```



```

        raise;
end WRITE_OUT;
--
procedure COPY_1 is
begin
    while not STUD_IO.END_OF_FILE(STUD_IN_1)
    loop
        READ_1;
        CURR_REC_OUT := CURR_REC_1_IN;
        WRITE_OUT;
    end loop;
    PUT(" First input file was last processed");
    -- Сюда можно поместить обработчик для тех
    -- исключительных ситуаций, которые будут
    -- распространяться (т.е. возбуждаться по-
    -- вторно) из процедур READ_1 или WRITE_OUT.
end COPY_1;
--
procedure COPY_2 is
begin
    while not STUD_IO.END_OF_FILE(STUD_IN_2)
    loop
        READ_2;
        CURR_REC_OUT := CURR_REC_2_IN;
        WRITE_OUT;
    end loop;
    PUT("Second input file was last processed");
    -- Здесь можно разместить обработчик исклю-
    -- чительных ситуаций (такой же, как в COPY_1).
end COPY_2;
--
procedure CLOSE_IT;
begin
    if STUD_IO.IS_OPEN(STUD_IN_1)
    then
        STUD_IO.CLOSE(STUD_IN_1);
    end if;
    if STUD_IO.IS_OPEN(STUD_IN_2)
    then
        STUD_IO.CLOSE(STUD_IN_2);
    end if;
    if STUD_IO.IS_OPEN(STUD_OUT_FILE)
    then
        STUD_IO.CLOSE(STUD_OUT_FILE);
    end if;
end CLOSE_IT;
--
begin
    -- Здесь удобно употребить оператор блока,
    -- так как он позволяет идентифицировать
    -- конкретный оператор, вызывающий возник-
    -- новение исключительной ситуации. В про-
    -- тивном случае будет неизвестно, в каком из
    -- операторов OPEN или CLOSE произошла
    -- ошибка.
begin
    STUD_IO.OPEN(FILE => STUD_IN_1,
                 MODE => IN_FILE,

```

```

NAME => "STUDENT-FILE-1-IN.DAT",
FORM => "" );
-- Используются принятые по умол-
-- чанию характеристики файла.
exception
when STUD_IO.STATUS_ERROR =>
-- Возможно, что файл уже открыт.
PUT(" Status error opening file " &
"STUDENT-FILE-1-IN.DAT");
raise;
-- Повторно возбуждается та же самая
-- исключительная ситуация
-- STATUS_ERROR. Она будет распростра-
-- няться далее (фактически, программа
-- завершится, а управление будет
-- передано во внешнюю среду).
-- Исключительная ситуация MODE_ERROR
-- здесь возникать не будет.
when STUD_IO.NAME_ERROR =>
-- Быть может, не существует внешний
-- файл, или же в строке, определяющей
-- параметр NAME, допущена ошибка.
PUT(" Name error opening file " &
"STUDENT-FILE-1-IN.DAT");
raise;
when STUD_IO.USE_ERROR =>
-- Возможно, задан неверный фактический
-- параметр, либо обнаружена иная не-
-- корректность.
PUT(" Use error opening file " &
"STUDENT-FILE-1-IN.DAT");
raise;
-- Ситуация END_ERROR не будет возникать
-- при открытии файла. Ситуация
-- DATA_ERROR невозможна.
when others =>
-- Прочие исключительные ситуации.
PUT(" Unusual kind of error opening file " &
"STUDENT-FILE-1-IN.DAT");
raise;
end;
begin
STUD_IO.OPEN(FILE => STUD_IN_2,
MODE => IN_FILE,
NAME => "STUDENT-FILE-2-IN.DAT",
FORM => "" );

exception
-- Исключительные ситуации в данном случае
-- будут те же, что и для файла STUD_IN_1.
when STUD_IO.STATUS_ERROR =>
PUT(" Status error opening file " &
"STUDENT-FILE-2-IN.DAT ");
raise;
when STUD_IO.NAME_ERROR =>
PUT(" Name error opening file " &
"STUDENT-FILE-2-IN.DAT ");
raise;

```

```

when STUD_IO.USE_ERROR =>
    PUT(" Use error opening file " &
        "STUDENT-FILE-2-IN.DAT ");
    raise;
when others =>
    PUT(" Unusual kind of error opening file " &
        "STUDENT-FILE-2-IN.DAT ");
    raise;
end;
begin
STUD_IO.CREATE(FILE => STUD_OUT_FILE,
                MODE => OUT_FILE,
                NAME => "STUD-MERGED-FILE.DAT",
                FORM => " ");

-- Создать объединенный файл.
exception
when STUD_IO.STATUS_ERROR =>
    -- Возможно, что файл уже открыт.
    PUT(" Status error creating the file " &
        "STUD-MERGED-FILE.DAT");
    raise;
when STUD_IO.MODE_ERROR =>
    -- Ошибка будет возникать, если создается
    -- файл с режимом обмена IN_FILE.
    PUT(" Mode error creating the file " &
        "STUD-MERGED-FILE.DAT");
    raise;
when STUD_IO.NAME_ERROR =>
    -- Быть может, в имени внешнего файла
    -- заданы недопустимые символы.
    PUT(" Name error creating the file " &
        "STUD-MERGED-FILE.DAT");
    raise;
when STUD_IO.USE_ERROR =>
    -- Возможно, неправильно задан фактиче-
    -- ский параметр FORM, либо допущена иная
    -- некорректность.
    PUT(" Use error creating the file " &
        "STUD-MERGED-FILE.DAT");
    raise;
-- Ситуация END_ERROR не может возникнуть
-- при создании файла.
-- Ошибка DATA_ERROR невозможна.
when others =>
    -- Любые другие исключительные ситуации.
    PUT(" Unusual kind of error creating file " &
        "STUD-MERGED-FILE.DAT");
    raise;
end;
-- Еще один блок, охватывающий основные действия.
begin
if not STUD_IO.END_OF_FILE(STUD_IN_1) and
not STUD_IO.END_OF_FILE(STUD_IN_2)
then
    READ_1;
    READ_2;
loop

```

```

if CURR_REC_1.IN.BIG_ST_ID <
  CURR_REC_2.IN.BIG_ST_ID
  then
    -- Запись данных из первого файла.
    CURR_REC_OUT := CURR_REC_1.IN;
    WRITE_OUT;
  if STUD_IO.END_OF_FILE(STUD_IN_1)
    then
      -- Записан последний элемент из пер-
      -- вого входного файла. Теперь запи-
      -- шем текущий элемент из второго
      -- входного файла. В противном слу-
      -- чае этот элемент будет отброшен в
      -- процедуре COPY_2. Слияние файлов
      -- заканчивается после окончания
      -- выполнения COPY_2.
      CURR_REC_OUT := CURR_REC_2.IN;
      WRITE_OUT;
      COPY_2;
      exit;
    end if;
  READ_1;
elseif CURR_REC_1.IN.BIG_ST_ID >
  CURR_REC_2.IN.BIG_ST_ID
  then
    -- Теперь выполняется запись данных
    -- из второго файла.
    CURR_REC_OUT := CURR_REC_2.IN;
    WRITE_OUT;
  if STUD_IO.END_OF_FILE(STUD_IN_2)
    then
      -- Записан последний элемент из вто-
      -- рого входного файла. Теперь запи-
      -- шем текущий элемент из первого
      -- входного файла. В противном слу-
      -- чае этот элемент будет отброшен в
      -- процедуре COPY_2. Слияние файлов
      -- заканчивается после окончания
      -- выполнения COPY_1.
      CURR_REC_OUT := CURR_REC_1.IN;
      WRITE_OUT;
      COPY_1;
      exit;
    end if;
  READ_2;
else
  -- Здесь складываются количества конт-
  -- рольных работ студентов.
  CURR_REC_OUT := CURR_REC_1.N;
  CURR_REC_OUT.NO_TESTS :=
    CURR_REC_OUT.NO_TESTS +
    CURR_REC_2.IN;
  CURR_REC_OUT.ST_TESTS
    (CURR_REC_1.IN.NO_TESTS + 1 ..
     CURR_REC_OUT.NO_TESTS) :=
    CURR_REC_2.IN.ST_TESTS

```

```

        (1 .. CURR_REC_2.IN.NO_TESTS);
-- Это - присваивание вырезки.
WRITE_OUT;
if STUD_IO.END_OF_FILE(STUD_IN_2)
then
    COPY_1;
    exit;
end if;
-- После того как достигнут конец одного
-- файла, копируем другой файл.
if STUD_IO.END_OF_FILE(STUD_IN_1)
then
    COPY_2;
    exit;
end if;
READ_1;
READ_2;
end if;
-- Здесь об'единяются два непустых файла.
-- По крайней мере один файл обработан.
-- Теперь скопируем оставшийся.
end loop;
elsif STUD_IO.END_OF_FILE(STUD_IN_1)
then
    COPY_2;
else
    -- Если управление попадает сюда, то
    -- файл STUD_IN_2 должен быть пустым.
    COPY_1;
end if;
PUT(" The merge is done ");
exception
    -- Эта исключительная ситуация возникнет
    -- в случае распространения исключительных
    -- ситуаций из процедур READ, WRITE, COPY
    -- и т.д.
    when others => CLOSE_IT;
end;
-- Конец блока, охватывающего основные
-- действия по обработке файлов.
CLOSE_IT;
-- Этот вызов выполняется при успешном заверше-
-- нии обработки.
end EXCEP_MERGE_PROC_GRADES;
```

### 11.3. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ ПРИ ПАРАЛЛЕЛЬНО ПРОТЕКАЮЩИХ ПРОЦЕССАХ

Как отмечалось в разд. 11.1, TASKING\_ERROR является предопределенной исключительной ситуацией, возникающей при некоторых событиях, происходящих во время взаимодействия задач. Это следующие события:

- невозможность запуска задачи;
- попытка установить связь с задачей, которая уже закончилась или находится в состоянии аварийного завершения;
- явное возбуждение состояния TASKING\_ERROR во время рандеву.

Исключительная ситуация **TASKING\_ERROR** возникает либо в вызывающей задаче в точке вызова (к примеру, когда вызываемая задача уже завершилась), либо в вызываемой задаче, если исключительная ситуация возникла во время рандеву.

Если исключительная ситуация возникла в вызванной задаче, то распространение ситуации будет зависеть от наличия локальных средств ее обработки. Если исключительная ситуация возбуждена в пределах оператора приема ассерта и для нее есть локальный обработчик, то она не должна распространяться. Впрочем, у программиста есть возможность повторно возбудить эту ситуацию и тем самым вызвать ее распространение. Если же локальный обработчик желаемой исключительной ситуации отсутствует, то эта ситуация распространяется на вызывающую задачу. Обратите внимание, что если вызывающая задача попадет в состояние аварийного завершения, то на вызванную задачу это никак не подействует, т. е. внутри ее не возбудится никакой исключительной ситуации.

Проиллюстрируем использование исключительных ситуаций, возникающих в задачах, на примере следующей программы, являющейся модификацией программы **PLANT\_SCHED** из гл. 10. Она видоизменена так, чтобы можно было учесть поломку любой из трех машин, занятых в производственном цикле. Будем считать, что если машина сломается, то ее нельзя починить для продолжения обработки данной партии деталей. Кроме того, для упрощения программы здесь опущены первые операторы цикла в задачах **MACH1**, **MACH2**, **MACH3** и **INSPECT**.

#### Программа **EXCEP\_PLANT\_SCHED**

```
with DISTRIBUTIONS; use DISTRIBUTIONS;
with CALENDAR; use CALENDAR;
procedure EXCEP_PLANT_SCHED is
  N_OF_FINISHED_PROD : NATURAL := 0;
  N_OF_REJECTS : NATURAL := 0;
  task COORDINATOR is
    entry SCHED;
  end COORDINATOR;
  task MACH1 is
    entry OPER1;
    entry DISABLE;
  end MACH1;
  task MACH2 is
    entry OPER2;
    entry DISABLE;
  end MACH2;
  task MACH3 is
    entry OPER1_N_2;
    entry DISABLE;
  end MACH3;
  task STOP_INSPECTION;
  task INSPECT is
    entry PRODUCT;
    entry TAKE_A_BREAK;
  end INSPECT;
  -- далее добавлена дополнительная задача,
  -- которая моделирует поломку машин MACH1,
  -- MACH2 или MACH3, если функция BREAK_DOWN
  -- дает соответственно значения 1, 2 или 3.
  -- Если же вырабатывается значение 4, то произ-
  -- водственный процесс продолжаться не может
  -- (например, сломаны MACH2 и MACH3).
  -- Любые другие числа означают, что текущие
```

```

-- СОСТОЯНИЯ МАШИН НЕ ИЗМЕНЯЮТСЯ.
task DISABLE_MACHINES;
--
task body MACH1 is
  MACH1_JOB_DURATION : DURATION;
begin
  MACH1_JOB_DURATION := PROBABILITY(TIME_MACH1);
  loop
    -- Эта часть программы добавлена для обра-
    -- ботки возможного вызова входа DISABLE.
    -- Идентичные тексты добавлены в задачи
    -- MACH2 и MACH3. Альтернативный способ
    -- реализации этой части:
    -- if DISABLE'COUNT > 0
    -- then
    --   accept DISABLE do
    --     loop null; end loop;
    --   end;
    -- end if;
    select
      accept DISABLE do
        loop
          null;
        end loop;
      end;
    else
      null;
    end select;
    -- Конец дополнительного текста.
    accept OPER1;
    delay MACH1_JOB_DURATION;
    -- Поместим вызов входа в блок.
    begin
      MACH2.OPER2;
    exception
      when TASKING_ERROR =>
        PUT(" Possibly terminated MACH2");
        raise;
    end;
    MACH1_JOB_DURATION := PROBABILITY(TIME_MACH1);
  end loop;
end MACH1;
--
task body MACH2 is
  MACH2_JOB_DURATION : DURATION;
begin
  MACH2_JOB_DURATION := PROBABILITY(TIME_MACH2);
  loop
    -- Начало дополнительного текста.
    select
      accept DISABLE do
        loop null; end loop;
      end;
    else null;
    end select;
    -- Конец дополнительного текста.
    accept OPER2;

```

```
delay MACH2_JOB_DURATION;
-- Поместим вызов задачи в блок.
begin
  INSPECT.PRODUCT;
  exception
    when TASKING_ERROR =>
      PUT(" Possibly terminated INSPECT");
      raise;
end;
MACH2_JOB_DURATION := PROBABILITY(TIME_MACH2);
end loop;
end MACH2;
--
task body MACH3 is
  MACH3_JOB_DURATION : DURATION;
begin
  MACH3_JOB_DURATION := PROBABILITY(TIME_MACH3);
  loop
    -- Начало дополнительного текста.
    select
      accept DISABLE do
        loop null; end loop;
      end;
    else null;
    end select;
    -- Конец дополнительного текста.
    accept OPER1_N_2;
    delay MACH3_JOB_DURATION;
    -- Поместим вызов задачи в блок.
    begin
      INSPECT.PRODUCT;
      exception
        when TASKING_ERROR =>
          PUT(" Possibly terminated INSPECT ");
          raise;
      end;
      MACH3_JOB_DURATION := PROBABILITY(TIME_MACH3);
    end loop;
  end MACH3;
--
task body COORDINATOR is
begin
  accept SCHED;
  loop
    loop
      select
        MACH1.OPER1;
      exit;
    or
      delay 0.01;
    end select;
    select
      MACH3.OPER1_N_2;
    exit;
    or
      delay 0.01;
    end select;
  end loop;
end loop;
```



```

        select;
        accept SCHED;
        else
            null;
        end select;
    end loop;
exception
    when TASKING_ERROR =>
        PUT(" Problem in one of the calls ");
        raise;
end COORDINATOR;
--
task body STOP_INSPECTION is
begin
    loop
        delay 120.00;
        INSPECT.TAKE_A_BREAK;
    end loop;
end STOP_INSPECTION;
--
task body INSPECT is
    INSP_PASSED : BOOLEAN;
    EXAM_TIME : DURATION;
    START_MANUFACT, STOP_MANUFACT : TIME;
    DURATION_MANUFACT : DURATION;
    package DURATION_IO is new FIXED_IO
        (DURATION);
    use DURATION_IO;
begin
    START_MANUFACT := CLOCK;
    INSP_PASSED := INSP_RESULT;
    EXAM_TIME := PROBABILITY(INSPECTION);
    loop
        accept PRODUCT;
        delay EXAM_TIME;
        if INSP_PASSED
            then
                N_OF_FINISHED_PROD := N_OF_FINISHED_PROD
                    + 1;
            else
                N_OF_REJECTS := N_OF_REJECTS + 1;
            end if;
        EXAM_TIME := PROBABILITY(INSPECTION);
        INSP_PASSED := INSP_RESULT;
        exit when N_OF_FINISHED_PROD = 200;
        if TAKE_A_BREAK'COUNT > 0
            then
                accept TAKE_A_BREAK;
                delay 15;
            end if;
    end loop;
    STOP_MANUFACT := CLOCK;
    DURATION_MANUFACT := START_MANUFACT -
        STOP_MANUFACT;
    PUT (DURATION_MANUFACT, 10, 2);
    abort STOP_INSPECTION;
-- Обратите внимание, что когда данная задача

```

```

-- закончится, то любое обращение к ней будет
-- приводить к возбуждению в вызывающей задаче
-- исключительной ситуации TASKING_ERROR. Си-
-- туация возникнет в точке вызова.
-- Поэтому в задаче MACH2 и позднее в зада-
-- чих MACH1, MACH3 и COORDINATOR возникнет
-- одна и та же исключительная ситуация
-- TASKING_ERROR.
end INSPECT;
-- Ниже располагается тело задачи DISABLE_MACHINES.
-- Каждую секунду (что соответствует одной минуте
-- в реальном производственном процессе) вызыва-
-- ется функция BREAK_DOWN (ПОЛОМКА). Спецификация
-- данной функции должна быть добавлена в пакет
-- DISTRIBUTIONS следующим образом:
-- function BREAK_DOWN return NATURAL;
-- Этот пакет должен содержать в составе своего
-- тела средства учета сломанных машин, чтобы
-- одна и та же машина не ломалась двукратно.
task body DISABLE_MACHINES;
  MAYBE_IS_BROKE : NATURAL;
begin
  loop;
    delay 1.0;
    MAYBE_IS_BROKE := BREAK_DOWN;
    case MAYBE_IS_BROKE is
      when 1      => MACH1.DISABLE;
      when 2      => MACH2.DISABLE;
      when 3      => MACH3.DISABLE;
      when 4      => abort MACH2;
                  abort MACH3;
                  -- Задача INSPECT может
                  -- еще быть активной.
                  abort INSPECT;
      when others => null;
    end case;
  end loop;
end DISABLE_MACHINES;
--
begin
  for I in 1 .. 200
    loop
      COORDINATOR.SCHED;
    end loop;
end EXCEP_PLANT_SCHED;

```

## УПРАЖНЕНИЯ

1. Перепишите заново программу EXCEP\_MERGE\_PROC\_GRADES, заменив процедуры READ\_1 и READ\_2 на новую процедуру с формальным параметром, который будет показывать, для какого файла следует выполнять чтение. Сделайте и другие необходимые модификации, чтобы должным образом идентифицировать возможные исключительные ситуации.

2. Перепишите заново каждую программу из гл. 5, включив в них соответствующие средства обработки исключительных ситуаций.

3. Видоизмените пакет BANK\_RESOURCES и программу BANK\_MAINT из гл. 8 так, чтобы включить в них адекватные средства обработки исключительных ситуаций.

# Приложения<sup>1)</sup>

## Приложение А

### ПРЕДОПРЕДЕЛЕННЫЕ АТТРИБУТЫ ЯЗЫКА

#### P'ADDRESS

Префикс Р обозначает объект, программный сегмент, метку или вход. Вырабатывается адрес первого блока памяти, отведенной под Р. Для подпрограммы, пакета, сегмента-задачи или метки это значение относится к машинному коду, связанному с соответствующим телом или оператором. Для входа это значение относится к точке соответствующего аппаратного прерывания. Значение данного атрибута принадлежит к типу ADDRESS, определенному в пакете SYSTEM (см. 13.7.2)<sup>2)</sup>

#### P'AFT

Префикс Р обозначает фиксированный подтип. Результатом запроса атрибута является значение, равное числу цифр после десятичной точки, необходимому для сохранения точности подтипа Р, если только значение шага delta этого подтипа Р не превышает 0.1. В противном случае получается единица. (P'AFT—это наименьшее положительное число N, для которого значение выражения  $(10 \cdot N) \cdot P'DELTA$  больше или равно единице.) Этот атрибут дает значение, относящееся к универсальному целому типу (см. 3.5.10)

#### P'BASE

Префикс Р обозначает тип или подтип. Значением этого атрибута служит тип, базовый для Р. Атрибут разрешается использовать только в качестве префикса другого атрибута, например P'BASE/FIRST (см. 3.3.3)

#### P'CALLABLE

Префикс Р соответствует типу «задача». Значение FALSE вырабатывается, если выполнение задачи Р закончено или полностью завершено или же если эта задача находится в аварийном состоянии. В противном случае получается значение TRUE. Тип значения атрибута—это предопределенный логический тип BOOLEAN (см. 9.9)

#### P'CONSTRAINED

Префикс Р обозначает объект, принадлежащий к типу с дискриминантами. Значение TRUE вырабатывается, если к объекту Р применяется уточнение дискриминанта или если объект является константой, включая случай, когда он выступает как формальный параметр или родовой формальный параметр с видом связи in. В противном случае получается значение FALSE. Если Р—родовой формальный параметр с видом связи in out или если Р—формальный параметр с видом связи in out или out, а обозначение типа, приведенное в соответствующей спецификации параметра, обозначает неуточненный тип с дискриминантами, то значение данного атрибута получается из значения атрибута соответствующего

<sup>1)</sup> Материал приложений перепечатан из Справочного руководства по языку Ада ANSI/MIL-STD-1815A. [Имеется перевод: Джехани Н. Язык Ада.—М.: Мир, 1988.]

<sup>2)</sup> Номера разделов соответствуют номерам разделов вышеприведенной книги.

	фактического параметра. Значение настоящего атрибута принадлежит к предопределенному логическому типу <b>BOOLEAN</b> (см. 3.7.4)
<b>P'CONSTRAINED</b>	Префикс <b>P</b> обозначает приватный тип или подтип. При запросе атрибута вырабатывается значение <b>FALSE</b> , если <b>P</b> обозначает неуточненный приватный тип с дискриминантами, не являющийся формальным параметром, а также если <b>P</b> обозначает родовой приватный тип, являющийся формальным параметром, а соответствующий фактический подтип — это либо неуточненный тип с дискриминантами, либо неуточненный регулярный тип. В остальных случаях получается значение <b>TRUE</b> . Значение данного атрибута принадлежит к предопределенному логическому типу <b>BOOLEAN</b> (см. 7.4.2)
<b>P'COUNT</b>	Префикс <b>P</b> обозначает вход или сегмент-задачу. При запросе атрибута вырабатывается значение, равное числу вызовов входа, находящихся в данный момент времени в очереди к нему. (Если этот атрибут вычисляется внутри оператора приема для входа <b>P</b> , то содержимое счетчика не включает вызов от вызывающей задачи.) Значение настоящего атрибута относится к <u>универсальному_целому</u> типу (см. 9.9)
<b>P'DELTA</b>	Префикс <b>P</b> обозначает фиксированный подтип. При запросе атрибута вырабатывается значение, равное <i>step delta</i> , указанному в определении абсолютной погрешности для подтипа <b>P</b> . Значение этого атрибута принадлежит к <u>универсальному_действительному</u> типу (см. 3.5.10)
<b>P'DIGITS</b>	Префикс <b>P</b> обозначает плавающий подтип. При запросе атрибута вырабатывается число, равное количеству десятичных цифр в мантиссе модельных чисел для подтипа <b>P</b> . (Этот атрибут дает число <b>D</b> из разд. 3.5.7 руководства по языку.) Значение данного атрибута принадлежит к <u>универсальному_целому</u> типу (см. 3.5.8)
<b>P'EMAX</b>	Префикс <b>P</b> обозначает плавающий подтип. При запросе атрибута вырабатывается число, равное наибольшему значению порядка числа в двоичном каноническом представлении модельных чисел для подтипа <b>P</b> . (Этот атрибут дает произведение $4 * B$ из разд. 3.5.7 руководства по языку.) Значение данного атрибута относится к <u>универсальному_целому</u> типу (см. 3.5.8)
<b>P'EPSILON</b>	Префикс <b>P</b> обозначает плавающий подтип. При запросе атрибута вырабатывается число, равное абсолютному значению разности между модельным числом 1.0 и следующим за ним большим модельным числом для подтипа <b>P</b> . Значение данного атрибута принадлежит к <u>универсальному_действительному</u> типу (см. 3.5.8)
<b>P'FIRST</b>	Префикс <b>P</b> обозначает скалярный тип или подтип скалярного типа. Атрибут дает нижнюю границу значений для <b>P</b> . Значение атрибута принадлежит к тому же типу, что и <b>P</b> (см. 3.5)
<b>P'FIRST</b>	Префикс <b>P</b> обозначает регулярный тип или уточненный регулярный подтип. Атрибут дает нижнюю границу диапазона значений для первого индекса. Значение атрибута имеет тот же тип, что и эта нижняя граница (см. 3.6.2 и 3.8.2)
<b>P'FIRST(N)</b>	Префикс <b>P</b> обозначает регулярный тип или уточненный регулярный подтип. Запрос атрибута дает нижнюю границу диапазона значений для <b>N</b> -го индекса. Значение атрибута имеет тот же тип, что и эта нижняя граница. Аргумент <b>N</b>

	должен быть статическим выражением <i>универсального_целого</i> типа. Значение N должно быть положительным (ненулевым) и не превышать размерности массива (см. 3.6.2 и 3.8.2)
P'FIRST_BIT	Префикс P обозначает компоненту структуры. Атрибут дает величину смещения первого бита компоненты, отсчитываемую от начала первого из блоков памяти, занимаемых компонентой. Величина смещения измеряется числом бит. Значение данного атрибута принадлежит к <i>универсальному_целому</i> типу (см. 13.7.2)
P'FORE	Префикс P обозначает фиксированный подтип. Запрос атрибута дает величину, равную минимальному числу символов, необходимому для целой части десятичного представления любого значения подтипа P. Данное представление не включает показатель степени, но включает односимвольный префикс, в качестве которого выступает знак минус или пробел. В этом минимальном числе не учитываются предшествующие нули и символы подчеркивания. Получается число, не меньшее двух. Значение данного атрибута принадлежит к <i>универсальному_целому</i> типу (см. 3.5.10)
P'IMAGE	Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией с одним параметром. Фактическим параметром X должна служить величина, относящаяся к базисному для P типу. Тип результата – предопределенный тип STRING. Результатом запроса атрибута является образ значения X, т. е. последовательность символов, представляющая значение X в формате, доступном для восприятия человеком. Образ целого числа – это соответствующий десятичный литерал без символов подчеркивания, без предшествующих нулей, без показателя степени и без последующих пробелов. Но это представление включает односимвольный префикс – знак минус или пробел. Образ перечисляемого значения – это либо соответствующий идентификатор, отображаемый на верхнем регистре, либо соответствующий односимвольный литерал, включающий два апострофа, без предшествующих и последующих пробелов. Образ символа, не входящего в набор графических знаков Ады, будет системно-зависимым (см. 3.5.5)
P'LARGE	Префикс P обозначает действительный подтип. Запрос атрибута дает наибольшее положительное модельное число для подтипа P. Значение данного атрибута относится к <i>универсальному_действительному</i> типу (см. 3.5.8 и 3.5.10)
P'LAST	Префикс P обозначает скалярный тип или подтип скалярного типа. Атрибут дает верхнюю границу значений для P. Значение данного атрибута принадлежит к тому же типу, что и P (см. 3.5)
P'LAST	Префикс P обозначает регулярный тип или уточненный регулярный подтип. Атрибут дает верхнюю границу диапазона значений первого индекса. Значение данного атрибута относится к тому же типу, что и эта верхняя граница (см. 3.6.2 и 3.8.2)
P'LAST(N)	Префикс P обозначает регулярный тип или уточненный регулярный подтип. Запрос атрибута дает верхнюю границу диапазона значений N-го индекса. Значение данного атрибута относится к тому же типу, что и эта верхняя граница. Аргументом N должно служить статическое выражение <i>универсального_целого</i> типа. Значение N должно быть положи-

тельным (ненулевым), оно не должно превышать количества измерений массива (см. 3.6.2 и 3.8.2)

**P'LAST\_BIT**

Префикс Р обозначает компоненту структуры. Атрибут дает величину смещения последнего бита компоненты относительно первого блока памяти, занимаемой этой компонентой. Величина смещения измеряется числом битов. Значение данного атрибута принадлежит к *универсальному\_целому* типу (см. 13.7.2)

**P'LENGTH**

Префикс Р обозначает регулярный тип или уточненный регулярный подтип. Атрибут дает величину, равную количеству значений, которое может принимать первый индекс (нуль в случае пустого диапазона значений индекса). Значение данного атрибута относится к *универсальному\_целому* типу (см. 3.6.2)

**P'LENGTH(N)**

Префикс Р обозначает регулярный тип или уточненный регулярный подтип. Атрибут дает величину, равную количеству значений, которое может принимать N-й индекс (нуль в случае пустого диапазона значений индекса). Значение данного атрибута относится к *универсальному\_целому* типу. Аргументом N должно служить статическое выражение *универсального\_целого* типа. Значение N должно быть строго положительным, оно не должно превышать количества измерений массива (см. 3.6.2 и 3.8.2)

**P'MACHINE\_EMAX**

Префикс Р обозначает плавающий тип или подтип. Атрибут дает наибольшее значение порядка в машинном представлении базового для Р типа. Значение данного атрибута относится к *универсальному\_целому* типу (см. 13.7.3)

**P'MACHINE\_EMIN**

Префикс Р обозначает плавающий тип или подтип. Атрибут дает наименьшее (максимальное по абсолютной величине) значение *порядка* в машинном представлении базового для Р типа. Значение данного атрибута принадлежит к *универсальному\_целому* типу (см. 13.7.3)

**P'MACHINE\_MANTISSA**

Префикс Р обозначает плавающий тип или подтип. Атрибут дает количество цифр *мантиссы* в машинном представлении базового для Р типа. (Здесь цифры – это расширенные цифры<sup>1)</sup> из диапазона от 0 до P'MACHINE\_RADIX – 1.) Значение данного атрибута принадлежит к *универсальному\_целому* типу (см. 13.7.3)

**P'MACHINE\_OVERFLOWS**

Префикс Р обозначает действительный тип или подтип. Атрибут дает значение TRUE, если каждая предопределенная операция над значениями базового для Р типа либо дает корректный результат, либо возбуждает исключительную ситуацию *NUMERIC\_ERROR* при переполнениях. В противном случае получится значение FALSE. Значения данного атрибута относятся к предопределенному типу *BOOLEAN* (см. 13.7.3)

**P'MACHINE\_RADIX**

Префикс Р обозначает плавающий тип или подтип. Атрибут дает значение *основания системы счисления*, которое используется в машинном представлении базисного для Р типа. Значение данного атрибута относится к *универсальному\_целому* типу (см. 13.7.3)

<sup>1)</sup> Под расширенными цифрами понимаются обычные цифры и буквенные обозначения цифр от 10 до 15 при основании системы счисления, большем десяти. – Прим. перев.

**P'MACHINE\_ROUNDS**

Префикс Р обозначает действительный тип или подтип. Атрибут дает значение TRUE, если в итоге каждой предопределенной арифметической операции над значениями базового для Р типа получается точный или округленный результат. В противном случае получается значение FALSE. Значение данного атрибута принадлежит к предопределенному типу BOOLEAN (см. 13.7.3)

**P'MANTISSA**

Префикс Р обозначает действительный подтип. Атрибут дает количество двоичных цифр в двоичной мантиссе модельных для подтипа Р чисел. (Этот атрибут вычисляет число В из разд. 3.5.9 для фиксированного типа.) Значение данного атрибута относится к *универсальному\_целому* типу (см. 3.5.8 и 3.5.10)

**P'POS**

Префикс Р обозначает дискретный тип или подтип. Этот атрибут является функцией одного параметра. Фактическим параметром Х должна быть величина базового для Р типа. Тип результата – *универсальный\_целый*. Результатом запроса атрибута является номер позиции значения фактического параметра в совокупности значений данного дискретного типа (см. 3.5.5)

**P'POSITION**

Префикс Р обозначает компонент структуры. Атрибут дает величину смещения первого из блоков памяти, занимаемых компонентой, относительно начала первого из блоков памяти, занимаемых структурой. Это смещение измеряется в блоках памяти. Значение данного атрибута относится к *универсальному\_целому* типу (см. 13.7.2)

**P'PRED**

Префикс Р обозначает дискретный тип или подтип. Этот атрибут является функцией одного параметра. Фактическим параметром Х должна быть величина базового для Р типа. Тип результата – это базовый для Р тип. Результатом запроса атрибута является значение, позиция которого в совокупности значений типа Р на единицу меньше, чем позиция Х. Если Х станет равным P'BASE/FIRST, то возникнет исключительная ситуация CONSTRAINT\_ERROR (см. 3.5.5)

**P'RANGE**

Префикс Р обозначает регулярный тип или уточненный регулярный подтип. Атрибут дает диапазон значений первого индекса Р, т.е. диапазон P'FIRST .. P'LAST (см. 3.6.2)

**P'RANGE(N)**

Префикс Р обозначает регулярный тип или подтип. Атрибут дает диапазон значений N-го индекса Р, т.е. диапазон P'FIRST(N) .. P'LAST (см. 3.6.2)

**P'SAFE\_EMAX**

Префикс Р обозначает плавающий тип или подтип. Атрибут дает наибольшее значение показателя степени в бинарной канонической форме «надежных» чисел базового для Р типа. (Этот атрибут вырабатывает число Е из разд. 3.5.7.) Значение данного атрибута принадлежит к *универсальному\_целому* типу (см. 3.5.8)

**P'SAFE\_LARGE**

Префикс Р обозначает действительный тип или подтип. Атрибут дает наибольшее «надежное» число базового для Р типа. Значение данного атрибута принадлежит к *универсальному\_действительному* типу (см. 3.5.8 и 3.5.10)

**P'SAFE\_SMALL**

Префикс Р обозначает действительный тип или подтип. Атрибут дает наименьшее строго положительное «надежное» число базового для Р типа. Значение данного атрибута принадлежит к *универсальному\_действительному* типу (см. 3.5.8 и 3.5.10)

P'SIZE	Префикс P обозначает объект. Атрибут выдает значение, равное количеству бит, выделенному для размещения объекта. Значение данного атрибута относится к <i>универсальному_целому</i> типу (см. 13.7.2)
P'SIZE	Префикс P обозначает любой тип или подтип. Атрибут дает значение, равное минимальному количеству бит, которое в данной системе потребуется выделить для размещения любого возможного объекта типа или подтипа P. Значение данного атрибута относится к <i>универсальному_целому</i> типу (см. 13.7.2)
P'SMALL	Префикс P обозначает действительный подтип. Атрибут дает наименьшее строго положительное число подтипа P. Значение данного атрибута относится к <i>универсальному_действительному</i> типу (см. 3.5.8 и 3.5.10)
P'SORAGE_SIZE	Префикс P обозначает ссылочный тип или подтип. Атрибут дает значение, равное общему количеству блоков памяти, зарезервированному под совокупность данных, ассоциируемую с базовым для P типом. Значение данного атрибута принадлежит к <i>универсальному_целому</i> типу (см. 13.7.2)
P'SORAGE_SIZE	Префикс P обозначает тип «задача» или объект этого типа. Атрибут дает значение, равное количеству блоков памяти, зарезервированному для каждой активации типа «задача» P или объекта «задача» P. Значение данного атрибута принадлежит к <i>универсальному_целому</i> типу (см. 13.7.2)
P'SUCC	Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией одного параметра. Фактическим параметром X должна служить величина базового для P типа. Тип результата – базовый для P тип. Результатом служит значение, позиция которого в совокупности значений типа P на единицу больше, чем у X. Если X будет равно P'BASE/LAST, то возникнет исключительная ситуация CONSTRAINT_ERROR (см. 3.5.5)
P'TERMINATED	Префикс P относится к типу «задача». Атрибут дает значение TRUE, если задача P окончательно завершилась. В противном случае получается значение FALSE. Значение данного атрибута принадлежит к предопределенному типу BOOLEAN (см. 9.9)
P'VAL	Префикс P обозначает дискретный тип или подтип. Этот атрибут является специальной функцией одного параметра X, который может принадлежать к любому целому типу. Тип результата – базовый для P тип. Результатом служит значение, номер позиции которого является величиной <i>универсального_целого</i> типа X. Если <i>универсальная_целая</i> величина X не будет находиться в диапазоне P'POS(P'BASE/FIRST) .. P'POS(P'BASE/LAST), то возникнет исключительная ситуация CONSTRAINT_ERROR (см. 3.5.5)
P'VALUE	Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией одного параметра. Фактический параметр X должен быть величиной предопределенного типа STRING. Тип результата – базовый для P тип. Все пробелы, стоящие перед или после последовательности символов, соответствующих X, игнорируются. Для перечисляемого типа: если последовательность символов имеет синтаксис перечисляемого литерала и если этот литерал существует для базового по отношению к P типу, что результатом будет соответствующее перечисляемое значение. Для целого типа:



если последовательность символов имеет синтаксис целого литерала с необязательным символом (плюс или минус), стоящим перед ним, и если существует соответствующее значение в множестве значений базового для Р типа, то результатом будет это значение. В любых других случаях будет возбуждено состояние `CONSTRAINT_ERROR` (см. 3.5.5)

#### **P'WIDTH**

Префикс Р обозначает дискретный подтип. Атрибут дает максимальную длину образа по всем значениям подтипа Р. (*Образ*—это последовательность символов, выдаваемая при запросе атрибута IMAGE.) Значение атрибута WIDTH принадлежит к универсальному\_целому типу (см. 3.5.5)

## Приложение Б

# ПРЕДОПРЕДЕЛЕННЫЕ ИНСТРУКЦИИ ДЛЯ ТРАНСЛЯТОРА ЯЗЫКА

В данном приложении определяются инструкции транслятору LIST, PAGE и OPTIMIZE и подытоживаются определения остальных инструкций языка Ада.

### Инструкция (pragma)

### Смысл инструкции транслятору

#### CONTROLLED

В качестве аргумента выступает простое имя ссылочного типа. Эту инструкцию можно располагать только непосредственно в декларативной части программы или в спецификации пакета, содержащих объявление ссылочного типа, причем объявление должно текстуально предшествовать данной инструкции. Не допускается использование инструкции для производных типов. Инструкция указывает транслятору, что для объектов, на которые указывают ссылочные значения, не должна выполняться автоматическая утилизация памяти, исключая выход из ближайшего оператора блока, тела подпрограммы или тела задачи, охватывающих объявление ссылочного типа, либо выход из главной программы (см. 4.8)<sup>1)</sup>

#### ELABORATE

В качестве аргументов используются одно или несколько простых имен, обозначающих библиотечные сегменты. Инструкция должна располагаться сразу после фразы подключения контекста для сегмента компиляции, т.е. перед следующим далее библиотечным или вторичным сегментом. Инструкция указывает транслятору, что соответствующее тело библиотечного сегмента должно быть обработано перед данным сегментом компиляции. Если данный сегмент компиляции является подсегментом, то тело библиотечного сегмента должно быть обработано перед обработкой другого тела библиотечного сегмента, породившего данный подсегмент (см. 10.5)

#### INLINE

В качестве аргументов используются одно или несколько имен, каждое из которых — это либо имя подпрограммы, либо имя родовой подпрограммы. Инструкция может располагаться либо на месте объявления в декларативной части программы или спецификации пакета, либо (при трансляции) после библиотечного сегмента, но перед любым последующим сегментом компиляции. Данная инструкция дает транслятору указание, чтобы объектный код тел подпрограмм вставлялся в код итоговой подпрограммы в любом месте их вызова, где это возможно<sup>2)</sup>. В случае родовой подпрограммы инструкция применяется к вызовам ее конкретизаций (см. 6.3.2)

<sup>1)</sup> См. сноски на стр. 300.

<sup>2)</sup> В этом случае принято говорить о так называемой открытой подпрограмме. — Прим. перев.

## INTERFACE

В качестве аргументов используются имя языка и имя подпрограммы. Инструкция должна располагаться на месте объявления и применяться к подпрограмме, объявленной ранее в той же декларативной части или спецификации пакета. Данную инструкцию также разрешается применять по отношению к библиотечному сегменту; в этом случае инструкция должна располагаться после объявления подпрограммы и перед любым последующим сегментом компиляции. Эта инструкция указывает название другого языка (и соответственно специфицирует необходимые соглашения о связях), а также информирует транслятор о том, что для соответствующей подпрограммы будет предоставлен объектный модуль (см. 13.9)

## LIST

Единственным аргументом служит один из идентификаторов ON или OFF. Эту инструкцию разрешается помещать в любом допустимом для инструкций месте программы. Инструкция LIST дает указание транслятору включить (ON) или выключить (OFF) печать листинга программы. Действие данной инструкции продолжается до тех пор, пока (в пределах одного и того же процесса компиляции) не встретится инструкция LIST с противоположным значением аргумента. Сама эта инструкция печатается всегда, когда компилятор генерирует листинг программы

## MEMORY\_SIZE

Единственным аргументом инструкции служит числовой литерал. Данную инструкцию можно размещать только в начале компилируемого текста программы, перед первым сегментом компиляции (если он есть) данного процесса трансляции. Инструкция предписывает использовать указанное значение числового литерала для определения именованного числа MEMORY\_SIZE (см. 13.7)

## OPTIMIZE

Единственным аргументом здесь служит один из идентификаторов TIME или SPACE. Эту инструкцию разрешается размещать только в декларативной части. Она относится ко всему блоку или телу, охватывающему эту декларативную часть. Инструкция указывает, что должно быть главным критерием оптимизации программы — время ее выполнения или объем занимаемой памяти

## PACK

Единственным аргументом служит имя комбинированного или регулярного типа. Правила расположения этой инструкции и ограничения для названного типа те же, что и для фазы представления. Эта инструкция указывает, что в качестве главного критерия при выборе способа представления заданного типа должна служить минимизация объема памяти (см. 13.1)

## PAGE

Эта инструкция не имеет аргумента и может располагаться в любом допустимом для инструкций месте программы. Она указывает, что текст программы, следующий после инструкции, должен печататься, начиная с новой страницы (если транслятор в это время генерирует листинг программы)

## PRIORITY

Единственным аргументом служит статическое выражение предопределенного целого типа PRIORITY. Данную инструкцию можно располагать только внутри спецификации сегмента-задачи или непосредственно в пределах декларативной части главной программы. Инструкция указывает приоритет конкретной задачи (или совокупности задач для типа «задача») или приоритет главной программы (см. 9.8)

## SHARED

Единственным аргументом служит простое имя переменной. Эту инструкцию можно использовать только для скалярной или ссылочной переменной, введенной путем объявления ее объекта. Объявление переменной и данная инструкция (именно в таком порядке) должны располагаться непосредственно в одной и той же декларативной части или спецификации пакета. Инструкция указывает, что каждая операция чтения или изменения заданной переменной служит и точкой синхронизации для этой переменной. В конкретной системе должен быть ограничен круг объектов, по отношению к которым данная инструкция может применяться. Этот круг должны составлять те объекты, для которых каждое непосредственное чтение и изменение значения реализованы как целостные (неделимые) операции (см. 9.11)

## STORAGE\_UNIT

Единственным аргументом служит числовой литерал. Данную инструкцию можно указывать только в начале компилируемой программы, перед первым сегментом компиляции (если он есть), участвующим в процессе трансляции. Использование инструкции приводит к использованию значения заданного числового литерала при определении именованного числа STORAGE\_UNIT (см. 13.7)

## SUPPRESS

В качестве аргументов используются идентификатор проверки и имя объекта, типа, подтипа, подпрограммы, сегмента-задачи или родового сегмента (второй аргумент может отсутствовать). Данную инструкцию разрешается располагать либо непосредственно в декларативной части, либо непосредственно в спецификации пакета. В последнем случае разрешается употреблять только такую форму инструкции, в которой имя обозначает ресурс (или несколько перекрытых подпрограмм), объявленный непосредственно в спецификации пакета. Разрешение отменить указанную проверку действует в пределах от места размещения инструкции до конца декларативной области, ассоциируемой с наиболее близким охватывающим<sup>1)</sup> оператором блока или программным сегментом. Если инструкция используется в спецификации пакета, то ее действие распространяется до конца области действия упомянутого ресурса. Если в инструкции указано имя, то разрешение отменить заданную проверку предстает в еще более ограниченном виде. Оно действует только для операций над названным объектом или над всеми объектами типа (базисного для названного типа или подтипа), для вызовов названной подпрограммы, для запусков задач названного типа «задача», для конкретизаций заданного родового сегмента (см. 11.7)

## SYSTEM\_NAME

Единственным аргументом служит перечисляемый литерал. Инструкцию можно размещать только в начале текста программы, подлежащей компиляции, перед первым сегментом компиляции (если он вообще есть), участвующим в данном процессе трансляции. Применение инструкции приводит к использованию перечисляемого литерала с заданным идентификатором для определения константы SYSTEM\_NAME. Настоящую инструкцию можно употреблять только в том случае, если указанный идентификатор соответствует одному из литералов типа NAME, объявленных в пакете SYSTEM (см. 13.7)

<sup>1)</sup> Имеется в виду наибольшая глубина вложения.—Прим. перев.

## Приложение В

### ПРЕДОПРЕДЕЛЕННОЕ ОКРУЖЕНИЕ ЯЗЫКА

В данном приложении описывается спецификация пакета STANDARD, содержащего все предопределенные идентификаторы языка. Соответствующее тело пакета будет системно-зависимым и оно здесь не показано.

Операции, являющиеся предопределенными для типов, объявленных в пакете STANDARD, даются здесь в виде комментариев, так как они объявлены неявно. Курсивом выделены псевдоимена анонимных типов (например, *универсальный\_целый*) и неопределенная информация (например, *системно\_зависимый* или *любой\_фиксированный\_тип*).

**package STANDARD is**

**type BOOLEAN is** (FALSE, TRUE);

-- Предопределенные операции отношений для этого типа:

```
-- function "=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "<" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function ">" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
```

-- Предопределенные логические операции и предопределенная операция логического отрицания:

```
-- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "or" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;  
-- function "not" (RIGHT : BOOLEAN) return BOOLEAN;
```

-- Универсальный тип *универсальный\_целый* является предопределенным.

**type INTEGER is** *системно\_зависимый*;

-- Предопределенные операции для этого типа:

```
-- function "=" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function "<" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function ">" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;  
-- function "+" (RIGHT : INTEGER) return INTEGER;  
-- function "-" (RIGHT : INTEGER) return INTEGER;  
-- function "abs" (RIGHT : INTEGER) return INTEGER;  
-- function "+" (LEFT, RIGHT : INTEGER) return INTEGER;  
-- function "-" (LEFT, RIGHT : INTEGER) return INTEGER;  
-- function "**" (LEFT, RIGHT : INTEGER) return INTEGER;
```

```
-- function "/" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "***" (LEFT : INTEGER; RIGHT : INTEGER) return
  INTEGER;
```

-- В конкретной реализации могут иметься добавочные предопределенные целые типы. Рекомендуется, чтобы имена таких типов заканчивались словом INTEGER, например SHORT\_INTEGER или LONG\_INTEGER. Спецификация каждой операции для типа *универсальный\_целый* или для любого другого добавочного предопределенного целого типа получается из спецификации соответствующей операции для типа INTEGER путем замены имени INTEGER на имя нужного типа. Исключением из этого правила служит операция возведения в степень, для которой типом правого операнда всегда остается тип INTEGER.

-- Тип *универсальный\_действительный* является предопределенным.

**type FLOAT is системно\_зависимый;**

-- Предопределенные операции для этого типа:

```
-- function "=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "+" (RIGHT : FLOAT) return FLOAT;
-- function "-" (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;
-- function "+" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "***" (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;
```

-- В конкретной реализации могут иметься добавочные предопределенные плавающие типы. Рекомендуется, чтобы названия таких типов заканчивались словом FLOAT, например SHORT\_FLOAT или LONG\_FLOAT. Спецификация каждой операции для типа *универсальный\_действительный* или для любого добавочного предопределенного плавающего типа получается из спецификации соответствующей операции для типа FLOAT путем замены слова FLOAT на имя нужного типа.

-- Для универсальных типов, кроме перечисленных выше операций, определены также и следующие:

```
-- function «*» (LEFT : универсальный_целый; RIGHT : универсальный_действительный) return
  универсальный_действительный;
-- function «*» (LEFT : универсальный_действительный; RIGHT : универсальный_целый) return
  универсальный_действительный;
-- function «/» (LEFT : универсальный_действительный; RIGHT : универсальный_целый) return
  универсальный_действительный;
-- Тип универсальный_фиксированный является предопределенным типом. Для него разрешены
  лишь такие операции:
-- function «*» (LEFT : любой_фиксированный_тип; RIGHT : любой_фиксированный_тип)
  return универсальный_фиксированный;
-- function «/» (LEFT : любой_фиксированный_тип; RIGHT : любой_фиксированный_тип)
  return универсальный_фиксированный;
```

-- Следующие символы входят в стандартный набор символов кода ASCII. Символьные литералы, соответствующие управляющим символам, не являются идентификаторами. В данном определении они выделены курсивом.

**type CHARACTER is**

( nul,	soh,	stx,	etx,	eot,	enq,	ack,	bel,
bs,	ht,	lf,	vt,	ff,	cr,	so,	si,
dle,	dc1,	dc2,	dc3,	dc4,	nak,	syn,	etb,
can,	em,	sub,	esc,	fs,	gs,	rs,	us,
' ',	'!',	'"',	'#',	'\$',	'%',	'&',	'"',
'(',	')',	'*',	'+',	',' ,	'-',	'.',	'/',
'0',	'1',	'2',	'3',	'4',	'5',	'6',	'7',
'8',	'9',	':',	';',	'<',	'=',	'>',	'?',
'@',	'A',	'B',	'C',	'D',	'E',	'F',	'G',
'H',	'I',	'J',	'K',	'L',	'M',	'N',	'O',
'P',	'Q',	'R',	'S',	'T',	'U',	'V',	'W',
'X',	'Y',	'Z',	'[',	']',	'^',	'_',	
'`',	'a',	'b',	'c',	'd',	'e',	'f',	'g',
'h',	'i',	'j',	'k',	'l',	'm',	'n',	'o',
'p',	'q',	'r',	's',	't',	'u',	'v',	'w',
'x',	'y',	'z',	'{',	' ',	'}',	'~',	del );

**for CHARACTER use** -- Сюда входят все 128 символов кода ASCII (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);

-- Предопределенные операции для типа CHARACTER, такие же, как и у любого перечисляемого  
-- типа.

**package ASCII is**

-- Управляющие символы:

NUL	: constant CHARACTER := nul;
STX	: constant CHARACTER := stx;
EOT	: constant CHARACTER := eot;
ACK	: constant CHARACTER := ack;
BS	: constant CHARACTER := bs;
LF	: constant CHARACTER := lf;
FF	: constant CHARACTER := ff;
SO	: constant CHARACTER := so;
DLE	: constant CHARACTER := dle;
DC2	: constant CHARACTER := dc2;
DC4	: constant CHARACTER := dc4;
SYN	: constant CHARACTER := syn;
CAN	: constant CHARACTER := can;
SUB	: constant CHARACTER := sub;
FS	: constant CHARACTER := fs;
RS	: constant CHARACTER := rs;
DEL	: constant CHARACTER := del;
SOH	: constant CHARACTER := soh;
ETX	: constant CHARACTER := etx;
ENQ	: constant CHARACTER := enq;
BEL	: constant CHARACTER := bel;
HT	: constant CHARACTER := ht;
VT	: constant CHARACTER := vt;
CR	: constant CHARACTER := cr;
SI	: constant CHARACTER := si;
DC1	: constant CHARACTER := dc1;
DC3	: constant CHARACTER := dc3;
NAK	: constant CHARACTER := nak;

```

ETB      : constant CHARACTER := etb;
EM       : constant CHARACTER := em;
ESC      : constant CHARACTER := esc;
GS       : constant CHARACTER := gs;
US       : constant CHARACTER := us;

```

-- Прочие символы:

```

EXCLAM   : constant CHARACTER := '!';
SHARP    : constant CHARACTER := '#';
PERCENT  : constant CHARACTER := '%';
COLON    : constant CHARACTER := ':';
QUERY    : constant CHARACTER := '?';
L_BRACKET : constant CHARACTER := '[';
R_BRACKET : constant CHARACTER := ']';
UNDERLINE : constant CHARACTER := '_';
L_BRACE  : constant CHARACTER := '{';
R_BRACE  : constant CHARACTER := '}';
QUOTATION : constant CHARACTER := '"';
DOLLAR   : constant CHARACTER := '$';
AMPERSAND : constant CHARACTER := '&';
SEMICOLON : constant CHARACTER := ';';
AT_SIGN  : constant CHARACTER := '@';
BACK_SLASH : constant CHARACTER := '\';
CIRCUMFLEX : constant CHARACTER := '^';
GRAVE    : constant CHARACTER := '`';
BAR      : constant CHARACTER := '|';
TILDE    : constant CHARACTER := '~';

```

-- Буквы нижнего регистра:

```
LC_A : constant CHARACTER := 'a';
```

```
...
```

```
LC_Z : constant CHARACTER := 'z';
```

```
end ASCII;
```

-- Предопределенные подтипы:

```
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
```

```
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

-- Предопределенный строковый тип:

```
type STRING is array (POSITIVE range < >) of CHARACTER;
pragma PACK (STRING);
```

-- Предопределенные операции для этого типа:

```

-- function "=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/"= (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;

-- function "&" (LEFT : STRING;      RIGHT : STRING)      return STRING;
-- function "&" (LEFT : CHARACTER;   RIGHT : STRING)      return STRING;
-- function "&" (LEFT : STRING;      RIGHT : CHARACTER)    return STRING;
-- function "&" (LEFT : CHARACTER;   RIGHT : CHARACTER)    return STRING;

```



**type DURATION is delta системно\_зависимо range системно\_зависимо;**

-- Предопределенные операции для типа DURATION, такие же, как и для любого другого  
-- фиксированного типа.

-- Предопределенные исключительные ситуации:

**CONSTRAINT\_ERROR : exception;**  
**NUMERIC\_ERROR : exception;**  
**PROGRAM\_ERROR : exception;**  
**STORAGE\_ERROR : exception;**  
**TASKING\_ERROR : exception;**

**end STANDARD;**

-- Пакет MACHINE\_CODE (если он есть) (см. 13.8)

-- Родовая процедура UNCHECKED\_DEALLOCATION (см. 13.10.1)

-- Родовая функция UNCHECKED\_DEALLOCATION (см. 13.10.1)

**package CALENDAR is**

**type TIME is private;**

**subtype YEAR\_NUMBER is INTEGER range 1901 .. 2099;**

**subtype MONTH\_NUMBER is INTEGER range 1 .. 12;**

**subtype DAY\_NUMBER is INTEGER range 1 .. 31;**

**subtype DAY\_DURATION is DURATION range 0.0 .. 86\_400.0;**

**function CLOCK return TIME;**

**function YEAR (DATE : TIME) return YEAR\_NUMBER;**

**function MONTH (DATE : TIME) return MONTH\_NUMBER;**

**function DAY (DATE : TIME) return DAY\_NUMBER;**

**function SECONDS (DATE : TIME) return DAY\_DURATION;**

**procedure SPLIT ( DATE : in TIME;**  
**YEAR : out YEAR\_NUMBER;**  
**MONTH : out MONTH\_NUMBER;**  
**DAY : out DAY\_NUMBER;**  
**SECONDS : out DAY\_DURATION);**

**function TIME\_OF ( YEAR : YEAR\_NUMBER ;**  
**MONTH : MONTH\_NUMBER;**  
**DAY : DAY\_NUMBER;**  
**SECONDS : DAY\_DURATION := 0.0) return**  
**TIME;**

**function "+" (LEFT : TIME; RIGHT : DURATION) return TIME;**

**function "+" (LEFT : DURATION; RIGHT : TIME) return TIME;**

**function "-" (LEFT : TIME; RIGHT : DURATION) return TIME;**

**function "-" (LEFT : TIME; RIGHT : TIME) return DURATION;**

**function "<" (LEFT, RIGHT : TIME) return BOOLEAN;**

**function "<=" (LEFT, RIGHT : TIME) return BOOLEAN;**

**function ">" (LEFT, RIGHT : TIME) return BOOLEAN;**

**function ">=" (LEFT, RIGHT : TIME) return BOOLEAN;**

**TIME\_ERROR : exception;**

-- Эта ситуация может быть возбуждена функциями TIME\_OF, «+» и «-»

**private**

-- Системно-зависимая реализация

**end;**

```

package SYSTEM is
  type ADDRESS is системно_зависимый;
  type NAME is системно_зависимый_перечисляемый_тип;
  SYSTEM_NAME : constant NAME := системно_зависимая;
  STORAGE_UNIT : constant := системно_зависимая;
  MEMORY_SIZE : constant := системно_зависимая;

  -- Системно-зависимые поименованные числа:
  MIN_INT : constant := системно_зависимая;
  MAX_INT : constant := системно_зависимая;
  MAX_DIGITS : constant := системно_зависимая;
  MAX_MANTISSA : constant := системно_зависимая;
  FINE_DELTA : constant := системно_зависимая;
  TICK : constant := системно_зависимая;

  -- Прочие системно-зависимые объявления
  subtype PRIORITY is INGENER range системно_зависимый;
  ...
end SYSTEM;

with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, OUT_FILE);

  -- Управление файлами
  procedure CREATE ( FILE : in out FILE_TYPE;
                     MODE : in FILE_MODE := OUT_FILE;
                     NAME : in STRING := "";
                     FORM : in STRING := " ");

  procedure OPEN ( FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
                   NAME : in STRING;
                   FORM : in STRING := " ");

  procedure CLOSE (FILE : in out FILE_TYPE);
  procedure DELETE (FILE : in out FILE_TYPE);
  procedure RESET (FILE : in out FILE_TYPE; MODE : in
                   FILE_MODE);
  procedure RESET (FILE : in out FILE_TYPE);

  function MODE (FILE : in FILE_TYPE) return FILE_MODE;
  function NAME (FILE : in FILE_TYPE) return STRING;
  function FORM (FILE : in FILE_TYPE) return STRING;
  function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

  -- Операции ввода и вывода
  procedure READ (FILE : in FILE_TYPE; ITEM : out
                  ELEMENT_TYPE);
  procedure WRITE (FILE : in FILE_TYPE; ITEM : in
                   ELEMENT_TYPE);

  function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

```

-- Исключительные ситуации

```
STATUS_ERROR : exception renames
               IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames
               IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames
               IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR     : exception renames
               IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames
               IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR     : exception renames
               IO_EXCEPTIONS.END_ERROR;
DATA_ERROR    : exception renames
               IO_EXCEPTIONS.DATA_ERROR;
```

private

-- Системно-зависимая реализация

end SEQUENTIAL\_IO;

with IO\_EXCEPTIONS;

generic

type ELEMENT\_TYPE is private;

package DIRECT\_IO is

type FILE\_TYPE is limited private;

type FILE\_MODE is (IN\_FILE, INOUT\_FILE, OUT\_FILE);

type COUNT is range 0 .. *системно\_зависимый*;

subtype POSITIVE\_COUNT is COUNT range 1 .. COUNT'LAST;

-- Управление файлами

```
procedure CREATE (FILE : in out FILE_TYPE;
                  MODE : in FILE_MODE := OUT_FILE;
                  NAME : in STRING := "";
                  FORM : in STRING := "");
```

```
procedure OPEN (FILE : in out FILE_TYPE;
                MODE : in FILE_MODE;
                NAME : in STRING;
                FORM : in STRING := "");
```

```
procedure CLOSE (FILE : in out FILE_TYPE);
```

```
procedure DELETE (FILE : in out FILE_TYPE);
```

```
procedure RESET (FILE : in out FILE_TYPE; MODE : in
                 FILE_MODE);
```

```
procedure RESET (FILE : in out FILE_TYPE);
```

```
function MODE (FILE : in FILE_TYPE) return FILE_MODE;
```

```
function NAME (FILE : in FILE_TYPE) return STRING;
```

```
function FORM (FILE : in FILE_TYPE) return STRING;
```

```
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
```

-- Операции ввода и вывода

```
procedure READ (FILE : in FILE_TYPE; ITEM : out
                ELEMENT_TYPE; FROM : POSITIVE_
                COUNT);
```

```
procedure READ (FILE : in FILE_TYPE; ITEM : out
                ELEMENT_TYPE);
```

```

procedure WRITE (FILE : in FILE_TYPE; ITEM : in
    ELEMENT_TYPE; TO : POSITIVE_COUNT);
procedure WRITE (FILE : in FILE_TYPE; ITEM : in
    ELEMENT_TYPE);

procedure SET_INDEX(FILE : in FILE_TYPE; TO : in
    POSITIVE_COUNT);

function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function SIZE (FILE : in FILE_TYPE) return COUNT;

function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;

-- Исключительные ситуации
STATUS_ERROR : exception renames
    IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR : exception renames
    IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR : exception renames
    IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR : exception renames
    IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames
    IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR : exception renames
    IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames
    IO_EXCEPTIONS.DATA_ERROR;

private
-- Системно-зависимая реализация
end DIRECT_IO;

with IO_EXCEPTIONS;
package TEXT_IO is
    type FILE_TYPE is limited private;
    type FILE_MODE is (IN_FILE, OUT_FILE);
    type COUNT is range 0 .. системно_зависимый;
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
    UNBOUNDED : constant COUNT := 0; -- размер строки и страницы
    subtype FIELD is INTEGER range 0 .. системно_зависимый;
    subtype NUMBER_BASE is INTEGER range 2 .. 16;
    type TYPE_SET is (LOWER_CASE, UPPER_CASE);

-- Управление файлами

procedure CREATE ( FILE : in out FILE_TYPE;
    MODE : in FILE_MODE := INOUT_FILE;
    NAME : in STRING := "";
    FORM : in STRING := "");

procedure OPEN ( FILE : in out FILE_TYPE;
    MODE : in FILE_MODE;
    NAME : in STRING;
    FORM : in STRING := "");

```

```

procedure CLOSE (FILE : in out FILE_TYPE);
procedure DELETE (FILE : in out FILE_TYPE);
procedure RESET (FILE : in out FILE_TYPE; MODE :
                  in FILE_MODE);
procedure RESET (FILE : in out FILE_TYPE);

function MODE (FILE : in FILE_TYPE) return FILE_MODE;
function NAME (FILE : in FILE_TYPE) return STRING;
function FORM (FILE : in FILE_TYPE) return STRING;
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

-- Установка входных и выходных файлов, принимаемых по умолчанию
procedure SET_INPUT (FILE : in FILE_TYPE);
procedure SET_OUTPUT (FILE : in FILE_TYPE);

function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;

function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;

-- Установка размеров строки и страницы
procedure SET_LINE_LENGTH (FILE : in FILE_TYPE; TO : in
                           COUNT);
procedure SET_LINE_LENGTH (TO : in COUNT);
procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE; TO : in
                           COUNT);
procedure SET_PAGE_LENGTH (TO : in COUNT);

function LINE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH return COUNT;

function PAGE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH return COUNT;

-- Управление номерами позиции в строке, строки и страницы

procedure NEW_LINE (FILE : in FILE_TYPE; SPACING : in
                   POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING : in POSITIVE_COUNT :=
                   1);

procedure SKIP_LINE (FILE : in FILE_TYPE; SPACING : in
                   POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING : in POSITIVE_COUNT :=
                   1);

function END_OF_LINE (FILE : in FILE_TYPE) return
                   BOOLEAN;
function END_OF_LINE return BOOLEAN;

procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE;

procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE;

function END_OF_PAGE (FILE : in FILE_TYPE) return
                   BOOLEAN;
function END_OF_PAGE return BOOLEAN;

```

```

function END_OF_FILE (FILE : in FILE_TYPE) return
    BOOLEAN;
function END_OF_FILE return BOOLEAN;
procedure SET_COL (FILE : in FILE_TYPE; TO : in
    POSITIVE_COUNT);
procedure SET_COL (TO : in POSITIVE_COUNT);
procedure SET_LINE (FILE : in FILE_TYPE; TO : in
    POSITIVE_COUNT);
procedure SET_LINE (TO : in POSITIVE_COUNT);
function COL (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL return POSITIVE_COUNT;
function LINE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;
function PAGE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;

```

-- Ввод-вывод символов

```

procedure GET(FILE : in FILE_TYPE; ITEM : out CHARACTER);
procedure GET(ITEM : out CHARACTER);
procedure PUT(FILE : in FILE_TYPE; ITEM : in CHARACTER);
procedure PUT(ITEM : in CHARACTER);

```

-- Ввод-вывод строк

```

procedure GET(FILE : in FILE_TYPE; ITEM : out STRING);
procedure GET(ITEM : out STRING);
procedure PUT(FILE : in FILE_TYPE; ITEM : in STRING);
procedure PUT(ITEM : in STRING);

```

```

procedure GET_LINE(FILE : in FILE_TYPE; ITEM : out STRING;
    LAST : out NATURAL);

```

```

procedure GET_LINE(ITEM : out STRING; LAST : out NATURAL);
procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in STRING);
procedure PUT_LINE(ITEM : in STRING);

```

-- Родовой пакет для ввода-вывода величин целых типов

```

generic
    type NUM is range <>;
package INTEGER_IO is
    DEFAULT_WIDTH : FIELD := NUM'WIDTH;
    DEFAULT_BASE : NUMBER_BASE := 10;
    procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH
        : in FIELD := 0);
    procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
    procedure PUT(FILE : in FILE_TYPE;
        ITEM : in NUM;
        WIDTH : in FIELD := DEFAULT_WIDTH;
        BASE : in NUMBER_BASE := DEFAULT_
            BASE);
    procedure PUT(ITEM : in NUM;
        WIDTH : in FIELD := DEFAULT_WIDTH;
        BASE : in NUMBER_BASE := DEFAULT_
            BASE);

```

```

procedure GET(FROM : in STRING; ITEM : out NUM; LAST :
               out POSITIVE);
procedure PUT(TO      : out STRING;
               ITEM     : in NUM;
               BASE     : in NUMBER_BASE := DEFAULT_
                           BASE);

```

**end** INTEGER\_IO;

-- Родовые пакеты для ввода-вывода величин действительных типов

**generic**

**type** NUM **is** digits <>;

**package** FLOAT\_IO **is**

```

  DEFAULT_FORE : FIELD := 2;
  DEFAULT_AFT  : FIELD := NUM'DIGITS-1;
  DEFAULT_EXP  : FIELD := 3;

  procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH :
                   in FIELD := 0);
  procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
  procedure PUT(FILE : in FILE_TYPE;
               ITEM  : in NUM;
               FORE  : in FIELD := DEFAULT_FORE;
               AFT   : in FIELD := DEFAULT_AFT;
               EXP   : in FIELD := DEFAULT_EXP);
  procedure PUT(ITEM : in NUM;
               FORE  : in FIELD := DEFAULT_FORE;
               AFT   : in FIELD := DEFAULT_AFT;
               EXP   : in FIELD := DEFAULT_EXP);

  procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out
                   POSITIVE);
  procedure PUT(TO      : out STRING;
               ITEM     : in NUM;
               AFT      : in FIELD := DEFAULT_AFT;
               EXP      : in FIELD := DEFAULT_EXP);

```

**end** FLOAT\_IO;

**generic**

**type** NUM **is** delta <>;

**package** FIXED\_IO **is**

```

  DEFAULT_FORE : FIELD := NUM'FORE;
  DEFAULT_AFT  : FIELD := NUM'AFT;
  DEFAULT_EXP  : FIELD := 0;

  procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH
                   : in FIELD := 0);
  procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
  procedure PUT(FILE : in FILE_TYPE;
               ITEM  : in NUM;
               FORE  : in FIELD := DEFAULT_FORE;
               AFT   : in FIELD := DEFAULT_AFT;
               EXP   : in FIELD := DEFAULT_EXP);
  procedure PUT(ITEM : in NUM;
               FORE  : in FIELD := DEFAULT_FORE;
               AFT   : in FIELD := DEFAULT_AFT;
               EXP   : in FIELD := DEFAULT_EXP);

```

```

procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out
    POSITIVE);
procedure PUT(TO      : out STRING;
    ITEM    : in NUM;
    AFT     : in FIELD := DEFAULT_AFT;
    EXP     : in FIELD := DEFAULT_EXP);

end FIXED_IO;

```

-- Родовой пакет для ввода-вывода величин перечисляемых типов

```

generic
  type ENUM is (<>);
  package ENUMERATION_IO is
    DEFAULT_WIDTH  : FIELD := 0;
    DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

    procedure GET(FILE : in FILE_TYPE; ITEM : out ENUM);
    procedure GET(ITEM : out ENUM);

    procedure PUT(FILE : in FILE_TYPE;
    ITEM : in ENUM;
    WIDTH : in FIELD := DEFAULT_WIDTH;
    SET : in TYPE_SET := DEFAULT_SETTING);

    procedure PUT(ITEM : in ENUM;
    WIDTH : in FIELD := DEFAULT_WIDTH;
    SET : in TYPE_SET := DEFAULT_SETTING);

    procedure GET(FROM : in STRING; ITEM : out ENUM; LAST : out
    POSITIVE);
    procedure PUT(TO      : out STRING;
    ITEM    : in ENUM;
    SET     : in TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

```

-- Исключительные ситуации

```

STATUS_ERROR : exception renames
    IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames
    IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames
    IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR     : exception renames
    IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames
    IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR     : exception renames
    IO_EXCEPTIONS.END_ERROR;
DATA_ERROR    : exception renames
    IO_EXCEPTIONS.DATA_ERROR;
LAYOUT_ERROR  : exception renames
    IO_EXCEPTIONS.LAYOUT_ERROR;

```

**private**

-- Системно-зависимая реализация  
**end** TEXT\_IO;



**package** IO\_EXCEPTIONS **is**

STATUS\_ERROR : exception;  
MODE\_ERROR : exception;  
NAME\_ERROR : exception;  
USE\_ERROR : exception;  
DEVICE\_ERROR : exception;  
END\_ERROR : exception;  
DATA\_ERROR : exception;  
LAYOUT\_ERROR : exception;

**end** IO\_EXCEPTIONS;

**package** LOW\_LEVEL\_IO **is**

-- Объявления возможных типов для DEVICE и DATA;

-- Объявления перекрытых процедур для этих типов:

**procedure** SEND\_CONTROL (DEVICE : *тип\_устройства*;  
DATA : *in out* *тип\_данных*);  
**procedure** RECEIVE\_CONTROL (DEVICE : *тип\_устройства*;  
DATA : *in out* *тип\_данных*);

**end**;

## Приложение Г

### СЛОВАРЬ ТЕРМИНОВ

Данное приложение не входит в состав стандартного определения языка программирования Ада. Термины, выделенные курсивом, либо присутствуют в словаре как самостоятельные ключевые слова, либо описаны при пояснении родственных терминов. После термина в скобках приведен соответствующий английский термин.

**Агрегат** (aggregate). В результате вычисления агрегата получается значение *составного типа*. Значение агрегата задается посредством указания значения каждой его *компоненты*. Для того чтобы показать, какое значение соответствует какой компоненте, можно воспользоваться либо *позиционным связыванием*, либо *поименованным связыванием*.

**Атрибут** (attribute). При запросе атрибута получается предопределенная характеристика заданного ресурса. Некоторые атрибуты являются *функциями*.

**Вариантная часть** (variant part) у *комбинированного типа* специфицирует его альтернативные *компоненты* и зависимости от значения дискриминанта. Каждое значение дискриминанта определяет выбор конкретной альтернативы из вариантной части.

**Вид связи** (mode). См. *параметр*.

**Видимая часть** (visible part). См. *пакет*.

**Видимость** (visibility). *Объявление* ресурса с некоторым идентификатором *видимо* в данной точке текста программы, если этот ресурс имеет приемлемый смысл для употребляемого в данном месте упомянутого идентификатора. Объявление будет *видимо посредством селекции* в месте расположения *селектора* при употреблении *селектируемой компоненты* или в месте расположения имени при *поименованном связывании*. Объявление будет *видимо непосредственно*, когда самостоятельно употребленный идентификатор будет иметь смысл, соответствующий этому объявлению.

**Возбуждение исключительной ситуации** (raising an exception). См. *исключительная ситуация*.

**Вход** (entry) используется для связи между *задачами*. Внешняя форма вызова входа в точности совпадает с вызовом *подпрограммы*. Внутреннее действие вызова входа специфицируется

одним или несколькими *операторами приема*, задающими действия, которые следует предпринять при вызове входа.

**Выражение** (expression) определяет вычисление значения.

**Вычисление** (evaluation) *выражения*—это процесс, посредством которого рассчитывается значение выражения. Данный процесс имеет место во время выполнения программы.

**Генератор** (allocator) создает *объект* и вырабатывает новое *ссылочное значение*, которое *указывает* на объект.

**Действительный тип** (real type)—*тип*, значения которого представляют собой аппроксимация действительных чисел. Существуют два вида действительных типов: фиксированные и плавающие. Для *фиксированных типов* указывается граница абсолютной погрешности представления, для *плавающих типов* указывается граница относительной погрешности представления, задаваемая в виде количества значащих десятичных цифр.

**Декларативная часть** (declarative part)—последовательность *объявлений*. Она может также содержать родственную информацию, такую, как *тела подпрограмм* и *фазы представления*.

**Диапазон** (range)—упорядоченный набор значений *скалярного типа*. Диапазон задается с помощью указания верхней и нижней границы значений. Значение из диапазона *принадлежит* этому диапазону.

**Дискретный тип** (discrete type)—*тип*, который имеет упорядоченный набор различающихся значений. К дискретным типам относятся *перечисляемые* и *целые типы*. Дискретные типы используются для индексации, для организации итераций, в альтернативах выбора в операторах

case и в вариантах выбора комбинированных типов с *вариантными частями*.

**Дискриминант** (discriminant) — особая *компонента объекта* или значения *комбинированного типа*. *Подтипы* остальных компонент и даже само их наличие или отсутствие могут зависеть от значения дискриминанта.

**Задача** (task) работает параллельно с остальными частями программы. Она состоит из *спецификации задачи* (в которой задается имя задачи, а также имена и *формальные параметры* ее входов) и *тела задачи* (в котором определяется порядок ее выполнения). *Сегмент-задача* — это один из видов *программных сегментов*. Тип «задача» — это тип, который позволяет в дальнейшем объявить любое количество сходных задач данного типа. Значение типа «задача» указывает на задачу.

**Имя** (name) — конструкция языка, которая выступает вместо ресурса. Имя обозначает ресурс, а ресурс является значением имени. См. также *объявление*, *префикс*.

**Индекс** (index). См. *регулярный тип*.

**Индексированная компонента** (indexed component) обозначает компоненту массива. Это — форма имени, содержащая выражения, которые указывают значения *индексов компоненты массива*. Индексированная компонента может также обозначать вход в семействе входов.

**Инструкция** (pragma) передает информацию транслятору.

**Исключительная ситуация** (exception) — сбойная ситуация, которая может возникнуть во время выполнения программы. *Возбуждение исключительной ситуации* приводит к прекращению нормального хода выполнения программы с тем, чтобы просигнализировать о возникновении ошибки. *Обработчик исключительной ситуации* — участок текста программы, в котором указываются действия, которые необходимо предпринять в качестве реакции на данную ситуацию. Выполнение такого участка текста программы называется *обработкой* исключительной ситуации.

**Квалифицированное выражение** (qualified expression) — *выражение*, перед которым стоит обозначение его типа или подтипа. Такая квалификация (уточнение) используется тогда, когда при ее отсутствии выражение могло бы оказаться неоднозначным (например, вследствие *перекрывтия*).

**Комбинированный тип** (record type). Значение комбинированного типа состоит из *компонент*, которые обычно имеют разные *типы* или *подтипы*. Для каждой компоненты значения комбинированного типа или *объекта* комбиниро-

ванного типа<sup>1)</sup> в определении этого типа указывается идентификатор, который однозначно определяет компонент структуры.

**Компонента** (component) — значение, которое является частью более сложного значения, или *объект*, который является частью более сложного объекта.

**Конкретный образец** (instance). См. *родовой сегмент*.

**Константа** (constant). См. *объект*.

**Лексический элемент** (lexical element) — идентификатор, *литерал*, разделитель или комментарий.

**Литерал** (literal) — значение, явно выраженное буквами, цифрами или прочими символами. Литерал — это либо числовой литерал, либо перечисляемый литерал, либо символьный литерал, либо строковый литерал.

**Модельное число** (model number) — точно представимое значение *действительного типа*. *Операции* для действительного типа определены в терминах операций над модельными числами этого типа. Совокупность свойств модельных чисел и операций над ними — это наименьшая совокупность свойств, сохраняемая всеми реализациями данного действительного типа.

**Непосредственная видимость** (direct visibility). См. *видимость*.

**Область действия** (scope). См. *объявление*.

**Обозначать** (denote). См. *объявление*.

**Обработка** (elaboration) *объявления* — это процесс, благодаря которому объявление достигает своей цели (например, создается *объект*). Этот процесс имеет место во время выполнения программы.

**Обработчик исключительной ситуации** (handler). См. *исключительная ситуация*.

**Объект** (object) содержит значение. В программе объект создается либо посредством *обработки объявления объекта*, либо при помощи *генератора*. В объявлении или в генераторе указывается *тип* объекта: объект может содержать значения только этого типа.

**Объявление** (declaration) связывает идентификатор (или иной вид обозначения) с ресурсом. Такое связывание действует в пределах части текста программы, называемой *областью действия объявления*. В пределах области действия объявления существуют участки, где можно использовать идентификатор для обращения к

<sup>1)</sup> В основном тексте книги объект комбинированного типа назван «структурой». — Прим. перев.

связанному с ним в объявлении значению. На таких участках идентификатор служит *простым именем* ресурса. Имя служит для обозначения связанного с ним ресурса.

**Объявление переименования** (renaming declaration) позволяет ввести добавочное имя для ресурса.

**Ограниченный тип** (limited type) — *тип*, для которого отсутствует неявное объявление операции присваивания и предопределенной операции проверки на равенство. Все типы «задачи» являются ограниченными. *Приватный тип* можно определить как ограниченный. Операцию проверки на равенство для ограниченного типа можно объявить явно.

**Оператор** (statement) специфицирует одно или более действий, которые должны быть произведены во время выполнения программы.

**Оператор блока** (block statement) — единичный оператор, который может содержать последовательность операторов. Он может также включать *декларативную часть и обработчики исключительных ситуаций*. Действие этих составных частей локализовано в операторе блока.

**Оператор приема** (accept statement). См. *вход*.

**Операция** в широком смысле (operation) — элементарное действие, связанное с одним или несколькими *типами*. Операция объявляется либо неявно — при объявлении типа, либо явно — при помощи *подпрограммы*, которая имеет *параметр* или *результат* объявленного типа.

**Операция** — унарная или бинарная<sup>1)</sup> (operator) — частный случай операции, которая имеет один или два операнда. Обозначение унарной операции записывается перед операндом. Обозначение бинарной операции записывается между операндами. Последняя форма операции является особым видом *вызова функции*. Операция может быть объявлена как *функция*. Многие операции декларируются неявно посредством *объявления типа*. Например, большинство объявлений типа подразумевает неявное объявление операции проверки на равенство для значений этого типа.

**Описание контекста** (context clause). См. *сегмент компиляции*.

**Пакет** (package). В пакете специфицируется группа логически связанных ресурсов, таких, как *типы*, *объекты* этих типов и *подпрограммы* с *параметрами* этих типов. Пакет делится на *объявление* и *тело пакета*. Объявление пакета

имеет *видимую часть*, содержащую *объявления* всех тех ресурсов, которые можно использовать явно за пределами данного пакета. В объявление пакета может также входить *приватная часть*, содержащая те детали структуры ресурсов, которые завершают спецификацию видимых ресурсов, но несущественны для пользователя пакета. *Тело пакета* содержит реализации *подпрограмм* (и, возможно, задач и других пакетов), которые были специфицированы в объявлении пакета. Пакет — это один из видов программных сегментов.

**Параметр** (parameter) — один из поименованных ресурсов, ассоциируемый с *подпрограммой*, *входом* или *родовым сегментом* и употребляемый для связи с соответствующим телом подпрограммы, *оператором приема* или родовым телом. **Формальный параметр** — идентификатор, используемый для обозначения именованного ресурса в пределах тела. **Фактический параметр** — это конкретный ресурс, связываемый с соответствующим формальным параметром при *вызове подпрограммы*, при *вызове входа* или в *родовой конкретизации*. Вид связи формального параметра определяет, будет ли связываемый фактический параметр задавать значение формального параметра, или же формальный параметр будет передавать значение фактическому параметру, или будет иметь место и то и другое. Связывание фактических параметров с формальными можно специфицировать при помощи *поименованных связываний*, *позиционных связываний* или их комбинаций.

**Перекрытие** (overloading). Один и тот же идентификатор может иметь в данной точке текста программы несколько различных значений. Это свойство называется *перекрытием*. Например, перекрытым перечисляемым литералом может быть идентификатор, появляющийся в определении двух и более перечисляемых типов. Фактический смысл перекрытого идентификатора определяется контекстом. *Подпрограммы*, *агрегаты*, *генераторы* и *строковые литералы* также могут быть перекрытыми.

**Переменная** (variable). См. *объект*.

**Перечисляемый тип** (enumeration type) — *дискретный тип*, значения которого представлены перечисляемыми литералами, которые задаются явно в *объявлении типа*. Перечисляемые литералы — это либо *идентификаторы*, либо *символьные литералы*.

**Плавающий тип** (floating point type). См. *действительный тип*.

**Подкомпонента** (subcomponent) — либо *компонента*, либо компонента другой подкомпоненты.

**Подпрограмма** (subprogram) — *процедура* или *функция*. Процедура определяет последова-

<sup>1)</sup> При переводе не делалось различия между операцией в широком смысле (operation) и бинарной или унарной операцией (operator) — они обе переводились как «операция». — Прим. перев.

тельность действий, она активизируется при помощи оператора *вызова процедуры*. Функция специфицирует последовательность действий и, кроме того, вырабатывает значение, называемое *результатом*. Таким образом, *вызов функции*—это выражение. Подпрограмма состоит из *объявления подпрограммы*, в котором указывается ее имя, *формальные параметры* и (для функции) тип результата, и *тела подпрограммы*, в котором задается последовательность действий. При вызове подпрограммы указываются *фактические параметры*, которые подлежат связыванию с формальными параметрами. Подпрограмма—это один из видов *программного сегмента*.

**Подсегмент (subunit)** См. *тело*.

**Подтип (subtype)** некоторого *типа* характеризует подмножество значений этого типа. Данное подмножество определяется *уточнением диапазона значений* этого типа. Каждое значение, входящее в набор значений подтипа, *принадлежит* этому подтипу и удовлетворяет уточнению, определяющему этот подтип.

**Позиционное связывание (positional association)** специфицирует связывание элемента с позицией в списке путем использования той же самой позиции для указания элемента.

**Поименованное связывание (named association)**. При поименованном связывании выполняется ассоциирование элемента с одной или более позиций списка, причем эти позиции называются по имени.

**Префикс (prefix)** используется в качестве первой части некоторых видов имен. Префикс—это либо *вызов функции*, либо *имя*.

**Приватная часть (private part)**. См. *пакет*.

**Приватный тип (private type)**—тип, структура и набор значений которого четко определены, но не видны непосредственно для пользователя этого типа. Приватный тип известен только по своим *дискриминантам* (если они есть) и по набору *операций*, определенных для него. Приватный тип и применимые для него операции определяются в *видимой части* пакета или в *родовой формальной части*. Для приватных типов также определены операции *присваивания* и проверки на равенство и неравенство, если только данный приватный тип не является *ограниченным*.

**Присваивание (assignment)**—операция, которая заменяет текущее значение *переменной* на новое значение. В левой части *оператора присваивания* указывается переменная, а в правой части—*выражение*, значение которого должно стать новым значением переменной.

**Программа (program)** состоит из некоторого числа *сегментов компиляции*, один из которых является *подпрограммой*, называемой *главной*

*программой*. Выполнение программы состоит из выполнения *главной программы*, которая может вызывать подпрограммы, объявленные в других сегментах компиляции программы.

**Программный сегмент (program unit)**—либо *родовой сегмент*, либо *пакет*, либо *подпрограмма*, либо *сегмент-задача*.

**Производный тип (derived type)**—тип, операции и значения которого являются копиями операций и значений существующего типа. Существующий тип называется *родительским типом* для данного производного типа.

**Простое имя (simple name)**. См. *объявление, имя*.

**Процедура (procedure)**. См. *подпрограмма*.

**Рандеву (rendezvous)**—взаимодействие между двумя параллельными *задачами*, происходящее в тот промежуток времени, когда одна задача вызвала *вход* другой задачи, а в вызванной задаче выполняется соответствующий *оператор приема* для вызывающей задачи.

**Регулярный тип (array type)**. Величина регулярного типа<sup>1)</sup> состоит из компонент, которые имеют один и тот же *подтип* (и, следовательно, принадлежат к одному и тому же типу). Каждая компонента однозначно идентифицируется своим *индексом* (для одномерного массива) или последовательностью индексов (для многомерного массива). Каждый индекс должен быть величиной *дискретного типа* и значение его должно лежать в пределах заданного *диапазона индексов*.

**Родительский тип (parent type)**. См. *производный тип*.

**Родовой сегмент (generic unit)**—трафарет-заготовка либо для набора *подпрограмм*, либо для набора *пакетов*. Подпрограмма или пакет, создаваемый при помощи этого трафарета, называется конкретизацией родového сегмента.

**Родовая конкретизация**—это разновидность *объявления*, которое создает конкретный образ ресурса. Родовой сегмент записывается в виде подпрограммы или пакета, перед спецификацией которых располагается префикс в виде *родовой формальной части*, в которой могут быть объявлены *родовые формальные параметры*. Родовой формальный параметр—это либо *тип*, либо *подпрограмма*, либо *объект*. Родовой сегмент—это один из видов *программных сегментов*.

**Сегмент компиляции (compilation unit)**—*объявление* или *тело программного сегмента*, представленное для компиляции в качестве независимого текста. Перед сегментом компиляции

<sup>1)</sup> Объект регулярного типа называется массивом.—Прим. перев.

может размещаться *описание контекста*, в котором перечисляются другие сегменты компиляции, от которых зависят данный сегмент. Это делается посредством одной или нескольких *фраз подключения контекста*.

**Селектируемая компонента** (selected component) имеет *имя*, состоящее из префикса и идентификатора, называемого *селектором*. Селектируемые компоненты используются для обозначения компонент структур, *входов*, и *объектов*, на которые указывают ссылочные значения. Они также используются в качестве *сословных имен*.

**Селектор** (selector). См. *селектируемая компонента*.

**Скалярный тип** (scalar type). *Объект* и значение скалярного *типа* не имеют *компонент*. Скалярный тип — это *дискретный тип* или *действительный тип*. Значения, принадлежащие к скалярному типу, упорядочены.

**Совокупность** (collection) — полный набор *объектов* для заданного *ссылочного типа*, созданных при помощи *генераторов*.

**Составное имя** (expanded name) обозначает ресурс, который *объявляется* непосредственно в пределах некоторой языковой конструкции. Составное имя записывается в форме с *селекцией компоненты*: *префикс* обозначает языковую конструкцию (*программный сегмент*, *блок*, *цикл* или *оператор приема*); *селектор* — это *простое имя ресурса*.

**Составной тип** (composite type) — тип, значения которого имеют *компоненты*. Существуют два вида составных типов — *регулярные типы* и *комбинированные типы*.

**Ссылочный тип** (access type) — набор значений *ссылочного типа* (*ссылочные значения*), которые могут быть либо *пустым (null) значением*, либо *значением, указывающим объект*, созданный *генератором*. Значение объекта можно считывать и изменять через *ссылочное значение*. При определении *ссылочного типа* специфицируется *тип объектов*, на которые могут указывать значения этого *ссылочного типа*. См. также *совокупность*.

**Тело (body)** — конструкция, определяющая порядок выполнения *подпрограммы*, *пакета* или *задачи*. *Заглушка (body stub)* — это форма тела, которая показывает, что порядок выполнения этого тела определяется в *раздельно компилируемом подсегменте*.

**Тип (type)** характеризует набор значений и набор *операций*, применимый к этим значениям. *Определение типа* — это языковая конструкция, которая определяет тип. Конкретный тип может быть *ссылочным*, *регулярным*, *приватным*, *комбинированным*, *скалярным типом* или *типом «задача»*.

**Удовлетворить (satisfy)**. См. *уточнение, подтип*.

**Указывать (designate)**. См. *ссылочный тип, задача*.

**Уточнение (constraint)** определяет подмножество значений для *типа*. Значение из этого подмножества удовлетворяет данному уточнению.

**Уточнение диапазона значений (range constraint)** для *типа* специфицирует диапазон и тем самым определяет подмножество значений *типа*, которое принадлежит этому диапазону.

**Уточнение диапазона индексов (index constraint)** для *регулярного типа* специфицирует верхние и нижние границы каждого *диапазона значений индексов* этого *регулярного типа*.

**Уточнение дискриминантов (discriminant constraint)** для *комбинированного типа* или *приватного типа* специфицирует значение каждого *дискриминанта* для этого *типа*.

**Фактический параметр (actual parameter)**. См. *параметр*.

**Фиксированный тип (fixed point type)**. См. *действительный тип*.

**Формальный параметр (formal parameter)**. См. *параметр*.

**Фраза использования (use clause)** обеспечивает *непосредственную видимость объявлений*, которые расположены в *видимых частях* перечисленных в этой фразе *пакетов*.

**Фраза подключения контекста (with clause)**. См. *сегмент компиляции*.

**Фраза представления (representation clause)** дает указание транслятору по поводу выбора *способа представления типа, объекта или задачи* в ЭВМ, которая выполняет программу. В некоторых случаях фразы представления полностью специфицируют этот способ. В других случаях они представляют критерии для выбора *способа представления*.

**Функция (function)**. См. *подпрограмма*.

**Целый тип (integer type)** — *дискретный тип*, значениями которого являются все целые числа из данного *диапазона*.

## Приложение Д

### СВОДКА СИНТАКСИСА

Данная сводка синтаксиса не является частью стандартного определения языка программирования Ада.

2.1<sup>1)</sup>

графический\_символ ::=  
 базисный\_графический\_символ  
 | буква\_нижнего\_регистра | прочий\_специальный\_символ

базисный\_графический\_символ ::=  
 буква\_верхнего\_регистра  
 | цифра | специальный\_символ | символ\_пробела  
 базисный\_символ ::= базисный\_графический\_символ  
 | символ\_управления\_форматом

2.3

идентификатор ::= буква {[символ\_подчеркивания]  
 буква\_или\_цифра}  
 буква\_или\_цифра ::= буква | цифра  
 буква ::= буква\_верхнего\_регистра  
 | буква\_нижнего\_регистра

2.4

десятичный\_литерал ::= десятичный\_литерал  
 | литерал\_с\_указанием\_системы\_счисления

2.4.1

десятичный\_литерал ::= целое\_число [целое\_число]  
 [порядок]  
 целое\_число ::= цифра {[символ\_подчеркивания]  
 цифра}  
 порядок ::= E [+ ] целое\_число | E - целое\_число

2.4.2

литерал\_с\_указанием\_системы\_счисления ::=  
 основание\_системы\_счисления #  
 целое\_число\_в\_указанной\_системе\_счисления  
 [целое\_число\_в\_указанной\_системе\_счисления] #  
 [порядок]

основание\_системы\_счисления ::= целое\_число  
 целое\_число\_в\_указанной\_системе\_счисления ::=  
 расширенная\_цифра {[символ\_подчеркивания]  
 расширенная\_цифра}  
 расширенная\_цифра ::= цифра | буква

2.5

символьный\_литерал ::= 'графический\_символ'

2.6

строковый\_литерал ::= "{графический\_символ}"

2.8

инструкция ::= **pragma** идентификатор

[(связывание\_аргумента {, связывание\_аргумента})];  
 связывание\_аргумента ::=  
 [идентификатор\_аргумента = >]  
 имя | [идентификатор\_аргумента = >] выражение

3.1.

базисное\_объявление ::=  
 | объявление\_объекта | объявление\_числа  
 | объявление\_типа | объявление\_подтипа  
 | объявление\_подпрограммы | объявление\_пакета  
 | объявление\_задачи | родовое\_объявление  
 | объявление\_исключительной\_ситуации  
 | родовая\_конкретизация | объявление\_переименования  
 | объявление\_отложенной\_константы

3.2

объявление\_объекта ::=  
 список\_идентификаторов : [constant]  
 указание : \_подтипа [:= выражение];  
 [список\_идентификаторов : [constant]  
 определение\_уточненного\_регуляторного\_типа  
 [:= выражение];  
 объявление\_числа ::= список\_идентификаторов :  
 constant := универсальное\_статическое\_выражение;  
 список\_идентификаторов ::=  
 идентификатор {, идентификатор}

3.3.1

объявление\_типа ::= полное\_объявление\_типа  
 | незавершенное\_объявление\_типа  
 | объявление\_приватного\_типа  
 полное\_объявление\_типа ::= **type** идентификатор  
 [дискриминантная\_часть] is определение\_типа;  
 определение\_типа ::=  
 определение\_перечисляемого\_типа  
 | определение\_целого\_типа  
 | определение\_действительного\_типа  
 | определение\_регулярного\_типа  
 | определение\_комбинированного\_типа  
 | определение\_ссылочного\_типа  
 | определение\_производного\_типа

3.3.2

объявление\_подтипа ::= **subtype**  
 идентификатор is указание\_подтипа;  
 указание\_подтипа ::= обозначение\_типа [уточнение]

<sup>1)</sup> См. сноски на с. 300.

обозначение\_типа ::= имя\_типа | имя\_подтипа

уточнение ::= уточнение\_диапазона\_значений  
| уточнение\_для\_плавающего\_типа  
| уточнение\_для\_фиксированного\_типа  
| уточнение\_диапазона\_индексов  
| уточнение\_дискриминантов

3.4

определение\_производного\_типа ::=  
new указание\_подтипа

3.5

уточнение\_диапазона\_значений ::=  
range диапазон\_значений  
диапазон\_значений ::= атрибут\_диапазона\_значений  
| простое\_выражение .. простое\_выражение

3.5.1

определение\_перечисляемого\_типа ::=  
(спецификация\_перечисляемого\_литерала  
{, спецификация\_перечисляемого\_литерала})  
спецификация\_перечисляемого\_литерала ::=  
перечисляемый\_литерал  
перечисляемый\_литерал ::= идентификатор  
| символьный\_литерал

3.5.4

определение\_целого\_типа ::=  
уточнение\_диапазона\_значений

3.5.6

определение\_действительного\_типа ::=  
уточнение\_для\_плавающего\_типа  
| уточнение\_для\_фиксированного\_типа

3.5.7

уточнение\_для\_плавающего\_типа ::=  
указание\_погрешности\_представления  
\_плавающего\_типа  
[уточнение\_диапазона\_значений]  
указание\_погрешности\_представления\_плавающего\_  
типа ::=  
digits статическое\_простое\_выражение

3.5.9

уточнение\_для\_фиксированного\_типа ::=  
указание\_погрешности\_представления\_  
фиксированного\_типа  
[уточнение\_диапазона\_значений]  
указание\_погрешности\_представления\_  
фиксированного\_типа ::=  
delta статическое\_простое\_выражение

3.6

определение\_регулярного\_типа ::=  
определение\_неуточненного\_регулярного\_типа  
| определение\_уточненного\_регулярного\_типа  
определение\_неуточненного\_регулярного\_типа ::=  
array (определение\_подтипа\_индекса  
{, определение\_подтипа\_индекса}) of  
указание\_подтипа\_компонент  
определение\_уточненного\_регулярного\_типа ::=  
array (уточнение\_индексов) of  
указание\_подтипа\_компонент  
определение\_подтипа\_индекса ::=  
обозначение\_типа range < >  
уточнение\_диапазонов\_индексов ::=  
(дискретный\_диапазон {, дискретный\_диапазон})  
дискретный\_диапазон ::=  
указание\_дискретного\_подтипа | диапазон

3.7

определение\_комбинированного\_типа ::=  
record  
  список\_компонент  
end record  
список\_компонент ::=  
объявление\_компоненты {объявление\_компоненты}  
| {объявление\_компоненты} вариантная\_часть  
| null;  
объявление\_компоненты ::=  
список\_идентификаторов :  
определение\_подтипа\_компонент [= выражение]  
определение\_подтипа\_компонент ::= указание\_  
подтипа

3.7.1

дискриминантная\_часть ::=  
(спецификация\_дискриминантов  
{, спецификация\_дискриминантов})  
спецификация\_дискриминантов ::=  
список\_идентификаторов :  
обозначение\_типа [= выражение]

3.7.2

уточнение\_дискриминантов ::=  
(связывание\_дискриминанта  
{, связывание\_дискриминанта})

связывание\_дискриминанта ::=  
[простое\_имя\_дискриминанта

3.7.3

{ простое\_имя\_дискриминанта} => выражение  
вариантная\_часть ::=  
case простое\_имя\_дискриминанта is  
  вариант  
  {вариант}  
end case;

вариант ::=

when условие\_выбора { | условие\_выбора} =>  
  список\_компонент  
условие\_выбора ::=  
  простое\_выражение | дискретный\_  
  диапазон | others | простое\_имя\_компоненты

3.8

определение\_ссылочного\_типа ::=  
access указание\_подтипа

3.8.1

незавершенное\_объявление\_типа ::= type  
  идентификатор [дискриминантная\_часть];

3.9

декларативная\_часть ::= {базисный\_элемент\_  
  объявления} {последующий\_элемент\_объявления}  
базисный\_элемент\_объявления ::=  
  базисное\_объявление  
  | фраза\_представления | фраза\_использования  
последующий\_элемент\_объявления ::= тело  
  | объявление\_подпрограммы | объявление\_пакета  
  | объявление\_задачи | родовое\_объявление  
  | фраза\_использования | родовая\_конкретизация  
тело ::= соответствующее\_тело | заглушка  
соответствующее\_тело ::= тело\_подпрограммы  
  | тело\_пакета | тело\_задачи

4.1

имя ::= простое\_имя | символьный\_литерал  
  | символ\_операции | я\_индексированная\_компонента  
  | вырезка | селектируемая\_компонента | атрибут



простое\_имя ::= идентификатор  
 префикс ::= имя | вызов функции

4.1.1  
 индексированная\_компонента ::=  
 префикс (выражение {, выражение})

4.1.2  
 вырезка ::= префикс (дискретный\_диапазон)

4.1.3  
 селектируемая\_компонента ::= префикс.селектор  
 селектор ::= простое\_имя | символьный\_литерал  
 символ\_операции | all

4.1.4  
 атрибут ::= префикс/указатель\_атрибута  
 указатель\_атрибута ::= простое\_имя  
 [(универсальное\_статическое\_выражение)]

4.3  
 агрегат ::= (связывание\_компоненты  
 {, связывание\_компоненты})  
 связывание\_компоненты ::= [условие\_выбора  
 { условие\_выбора } = > ] выражение

4.4  
 выражение ::=  
 отношение {and отношение}  
 | отношение {and then отношение}  
 | отношение {or отношение}  
 | отношение {or else отношение}  
 | отношение [xor отношение]  
 отношение ::=  
 простое\_выражение [операция\_отношения  
 простое\_выражение]  
 | простое\_выражение [not] in диапазон  
 | простое\_выражение [not] in обозначение\_типа

простое\_выражение ::=  
 [унарная\_аддитивная\_операция]  
 терм {бинарная\_аддитивная\_операция терм}  
 терм ::= множитель {мультипликативная\_операция  
 множитель}  
 множитель ::= простейшее\_выражение [**\*\***  
 простейшее\_выражение]  
 | abs простейшее\_выражение | not простейшее  
 выражение

простейшее\_выражение ::= числовой\_литерал | null  
 | агрегат | строковый\_литерал | имя | генератор  
 | вызов\_функции | преобразование\_типа  
 | квалифицированное\_выражение | (выражение)

4.5  
 логическая\_операция ::= and | or | xor  
 операция\_отношения ::= = | /= | < | <= | > | >=  
 бинарная\_аддитивная\_операция ::= + | - | &  
 унарная\_аддитивная\_операция ::= + | -  
 мультипликативная\_операция ::= \* | / | mod | rem  
 операция\_наивысшего\_старшинства ::= **\*\*** | abs | not

4.6  
 преобразование\_типа ::=  
 обозначение\_типа (выражение)

4.7  
 квалифицированное\_выражение ::=  
 обозначение\_типа(выражение)  
 | обозначение\_типа агрегат

4.8  
 генератор ::= new указание\_подтипа  
 | new квалифицированное\_выражение

5.1  
 последовательность\_операторов ::=  
 оператор {оператор}  
 оператор ::= {метка} простой\_оператор  
 | {метка} составной\_оператор  
 простой\_оператор ::= пустой\_оператор  
 | оператор\_присваивания | оператор\_вызова\_  
 процедуры  
 | оператор\_выхода | оператор\_возврата | оператор  
 перехода  
 | оператор\_вызова\_входа | оператор\_задержки  
 | оператор\_возбуждения\_исключительной\_ситуации  
 | оператор\_прекращения\_задачи  
 | оператор\_вставки\_кода  
 составной\_оператор ::=  
 | условный\_оператор | оператор\_выбора  
 | оператор\_цикла | оператор\_блока  
 | оператор\_приема | оператор\_отбора  
 метка ::= «простое\_имя\_метки»  
 пустой\_оператор ::= null;

5.2  
 оператор\_присваивания ::=  
 имя\_переменной ::= выражение;

5.3  
 условный\_оператор ::=  
 if условие then  
 последовательность\_операторов  
 {elsif условие then  
 последовательность\_операторов}  
 [else  
 последовательность\_операторов]  
 end if;  
 условие ::= логическое\_выражение

5.4  
 оператор\_выбора ::=  
 case выражение is  
 альтернатива\_оператора\_выбора  
 {альтернатива\_оператора\_выбора}  
 end case;  
 альтернатива\_оператора\_выбора ::=  
 when условие\_выбора { | условие\_выбора }  
 = > последовательность\_операторов

5.5  
 оператор\_цикла ::=  
 [простое\_имя\_цикла:] [схема\_итераций]  
 loop последовательность\_операторов  
 end loop [простое\_имя\_цикла];  
 схема\_итераций ::= while условие  
 | for спецификация\_параметров\_цикла  
 спецификация\_параметров\_цикла ::= идентификатор  
 in [reverse] дискретный\_диапазон

5.6  
 оператор\_блока ::=  
 [простое\_имя\_блока:]  
 [declare  
 декларативная\_часть]  
 begin  
 последовательность\_операторов  
 [exception

```

обработчик_исключительной_ситуации
{обработчик_исключительной_ситуации}}
end [простое_имя_блока]

```

5.7

```

оператор_выхода ::=
exit [имя_цикла] [when условие];

```

5.8

```

оператор_возврата ::= return [выражение];

```

5.9

```

оператор_перехода ::= goto имя_метки;

```

6.1

```

объявление_подпрограммы ::=
спецификация_подпрограммы;
спецификация_подпрограммы ::=
procedure идентификатор [формальная_часть]
| function обозначение [формальная_часть]
return обозначение_типа

```

```

обозначение ::= идентификатор | символ_операции
символ_операции ::= строковый_литерал
формальная_часть ::= (спецификация_параметров
{; спецификация_параметров})

```

```

спецификация_параметров ::=

```

```

список_идентификаторов:
вид_связи обозначение_типа [= выражение]
вид_связи ::= [in] | in out | out

```

6.3

```

тело_подпрограммы ::=
спецификация_подпрограммы is
[декларативная_часть]
begin
последовательность_операторов
[exception
обработчик_исключительной_ситуации
{обработчик_исключительной_ситуации}}
end [обозначение];

```

6.4

```

оператор_вызова_процедуры ::= имя_процедуры
[часть_с_фактическими_параметрами];
вызов_функции ::= имя_функции
[часть_с_фактическими_параметрами]
часть_с_фактическими_параметрами ::=
(связывание_параметра {, связывание_параметра})
связывание_параметра ::= [формальный_параметр
=>]

```

```

фактический_параметр
формальный_параметр ::= простое_имя_параметра
фактический_параметр ::= выражение
| имя_переменной
| обозначение_типа (имя_переменной)

```

7.1

```

объявление_пакета ::= спецификация_пакета;
спецификация_пакета ::=
package идентификатор is
{базисный_элемент_объявления}
private
{базисный_элемент_объявления}
end [простое_имя_пакета]
тело_пакета ::=
package body простое_имя_пакета is
[декларативная_часть]
begin
последовательность_операторов

```

```

[exception
обработчик_исключительной_ситуации
{обработчик_исключительной_ситуации}}]
end [простое_имя_пакета];

```

7.4

```

объявление_приватного_типа ::= type
идентификатор [дискриминантная_часть]
is [limited] private;
объявление_отложенной_константы ::= список_
идентификаторов : constant обозначение_типа;

```

8.4

```

фраза_использования ::= use
имя_пакета {, имя_пакета}

```

8.5

```

объявление_переименования ::= идентификатор :
обозначение_типа renames имя_объекта;
| идентификатор : exception renames
имя_исключительной_ситуации
| package идентификатор renames имя_пакета;
| спецификация_подпрограммы renames
имя_подпрограммы_или_входа;

```

9.1

```

объявление_задачи ::= спецификация_задачи;
спецификация_задачи ::=
task [type] идентификатор [is
{объявление_входа}
{фраза_представления}
end [простое_имя_задачи]]

```

```

тело_задачи ::=

```

```

task body простое_имя_задачи is
[декларативная_часть]
begin
последовательность_операторов
[exception
обработчик_исключительной_ситуации
{обработчик_исключительной_ситуации}}]
end [простое_имя_задачи];

```

9.5

```

объявление_входа ::=
entry идентификатор [(дискретный_
диапазон)] [формальная_часть];
оператор_вызова_входа ::= имя_входа
[часть_с_фактическими_параметрами];
оператор_приема ::= accept
простое_имя_входа [(индекс_входа)]
[формальная_часть] [do последовательность_
операторов
end [простое_имя_входа];
индекс_входа ::= выражение

```

9.6

```

оператор_задержки ::= delay простое_выражение;

```

9.7

```

оператор_отбора ::= селективное_ожидание
| условный_вызов_входа
| таймированный_вызов_входа

```

9.7.1

```

селективное_ожидание ::=
select
альтернатива_отбора
{or
альтернатива_отбора}
[else

```

```

последовательность_операторов]
end select;
альтернатива_отбора ::=
[when условие =>]
альтернатива_селективного_ожидания
альтернатива_селективного_ожидания ::=
альтернатива_приема | альтернатива_с_задержкой
| терминирующая_альтернатива
альтернатива_приема ::= оператор_приема
[последовательность_операторов]
альтернатива_задержки ::= оператор_задержки
[последовательность_операторов]
терминирующая_альтернатива ::= terminate;

```

## 9.7.2

```

условный_вызов_входа ::=
select
оператор_вызова_входа
[последовательность_операторов]
else
последовательность_операторов
end select;

```

## 9.7.3

```

таймированный_вызов_входа ::=
select
оператор_вызова_входа
[последовательность_операторов]
or
альтернатива_с_задержкой
end select;

```

## 9.10

```

оператор_прекращения_задачи ::= abort
имя_задачи {, имя_задачи}

```

## 10.1

```

компиляция ::= {сегмент_компиляции}
сегмент_компиляции ::=
описание_контекста библиотечный_сегмент
| описание_контекста вторичный_сегмент
библиотечный_сегмент ::=
объявление_подпрограммы
| объявление_пакета | родовое_объявление
| родовая_конкретизация | тело_подпрограммы
вторичный_сегмент ::=
тело_библиотечного_сегмента | подсегмент
тело_библиотечного_сегмента ::=
тело_подпрограммы | тело_пакета

```

## 10.1.1

```

описание_контекста ::= {фраза_
подключения_контекста {фраза_использования}}
фраза_подключения_контекста ::= with
простое_имя_сегмента {, простое_имя_сегмента};

```

## 10.2

```

заглушка ::=
спецификация_подпрограммы is separate;
| package body
простое_имя_пакета is separate;
| task body
простое_имя_задачи is separate;
подсегмент ::= separate

```

(имя\_порождающего\_сегмента) соответствующее  
тело

## 11.1

```

объявление_исключительной_ситуации ::=
список_идентификаторов : exception;

```

## 11.2

```

обработчик_исключительной_ситуации ::= when
условие_выбора_исключительной_ситуации
{ | условие_выбора_исключительной_ситуации }
=> последовательность_операторов
условие_выбора_исключительной_ситуации ::=
имя_исключительной_ситуации | others

```

## 11.3

```

оператор_возбуждения_исключительной_ситуации ::=
raise [имя_исключительной_ситуации];

```

## 12.1

```

родовое_объявление ::= родовая_спецификация;
родовая_спецификация ::= родовая_формальная_
часть спецификация_подпрограммы
| родовая_формальная_часть спецификация_пакета
родовая_формальная_часть ::= generic
{объявление_родовых_параметров}
объявление_родовых_параметров ::=
список_идентификаторов : [in [out]]
обозначение_типа [= выражение]
| type идентификатор is родовое_определение_типа;
| объявление_приватного_типа
| with спецификация_подпрограммы [is имя];
with спецификация_подпрограммы [is < >];
родовое_определение_типа ::= (< >) range < >
| digits < > | delta < >
| определение_регулярного_типа
| определение_ссылочного_типа

```

## 12.3

```

родовая_конкретизация ::=
package идентификатор is
new имя_родового_пакета
[родовая_фактическая_часть];
| procedure идентификатор is
new имя_родовой_процедуры
[родовая_фактическая_часть];
| function обозначение is
new имя_родовой_функции
[родовая_фактическая_часть];
родовая_фактическая_часть ::= (родовое_связывание
{, родовое_связывание})
родовое_связывание ::= [родовой_формальный_
параметр = >]

```

```

родовой_фактический_параметр
родовой_формальный_параметр ::=
простое_имя_параметра
| символ_операции
родовой_фактический_параметр ::= выражение
| имя_переменной | имя_подпрограммы
| имя_входа | обозначение_типа

```

## 13.1

```

фраза_представления ::= фраза_представления_типа
| адресная_фраза

```

фраза\_представления\_типа ::= фраза\_длины  
 | фраза\_представления\_перечисляемого\_типа  
 | фраза\_представления\_комбинированного\_типа

13.2

фраза\_длины ::= **for** атрибут **use**  
 простое\_выражение;

13.3

фраза\_представления\_перечисляемого\_типа ::=  
**for** простое\_имя\_типа **use** агрегат;

13.4

фраза\_представления\_комбинированного\_типа ::=  
**for** простое\_имя\_типа **use**  
**record** [фраза\_выравнивания]

{фраза\_представления\_компоненты}  
**end record**;

фраза\_выравнивания ::= **at mod**

простое\_статическое\_выражение;

фраза\_представления\_компоненты ::=  
 имя\_компоненты

**at** простое\_статическое\_выражение

**range** статический\_диапазон;

13.5

адресная\_фраза ::= **for** простое\_имя  
**use at** простое\_выражение;

13.8

оператор\_включения\_кода ::=  
 обозначение\_типа'агрегат-структура;

## ССЫЛКИ

Ниже перечислены названия синтаксических категорий и указан номер раздела, где эти категории определены <sup>1)</sup>. Например:

**операция\_сложения** 4.5

Кроме того, вслед за названием каждой синтаксической категории располагаются названия других категорий, в определениях которых она упоминается. Например, операция сложения упоминается в определении простого выражения:

**операция\_сложения** 4.5

**простое\_выражение** 4.4

Многоточие обозначает, что данная синтаксическая категория не определяется синтаксическими правилами. Например:

**буква\_нижнего\_регистра** ...

Все случаи употребления скобок даны после термина «( )». Префиксы, выделенные курсивом, которые были использованы выше с некоторыми терминами, здесь удалены.

<b>агрегат</b>	4.3	<b>базисный_символ</b>	2.1
квалифицированное_выражение	4.7	<b>базисный_элемент_объявления</b>	3.9
оператор_включения_кода	13.8	декларативная_часть	3.9
простейшее_выражение	4.4	спецификация_пакета	7.1
фраза_представления_перечисляемого_типа	13.3	<b>библиотечный_сегмент</b>	10.1
<b>адресная_фаза</b>	13.5	сегмент_компиляции	10.1
фаза_представления	13.1	<b>бинарная_аддитивная_операция</b>	4.5
<b>альтернатива_оператора_выбора</b>	5.4	простое_выражение	4.4
оператор_выбора	5.4	<b>буква</b>	2.3
<b>альтернатива_отбора</b>	9.7.1	расширенная_цифра	2.4.2
селективное_ожидание	9.7.1	идентификатор	2.3
<b>альтернатива_селективного_ожидания</b>	9.7.1	буква_или_цифра	2.3
альтернатива_отбора	9.7.1	<b>буква_верхнего_регистра</b>	...
<b>альтернатива_с_задержкой</b>	9.7.1	базисный_графический_символ	2.1
альтернатива_селективного_ожидания	9.7.1	буква	2.3
таймированный_вызов_входа	9.7.3	<b>буква_или_цифра</b>	2.3
<b>альтернатива_привода</b>	9.7.1	идентификатор	2.3
альтернатива_селективного_ожидания	9.7.1	<b>буква_нижнего_регистра</b>	...
<b>атрибут</b>	4.1.4	графический_символ	2.1
диапазон_значений	3.5	буква	2.3
имя	4.1	<b>вариант</b>	3.7.3
фраза_длины	13.2	вариантная_часть	3.7.3
<b>базисный_графический_символ</b>	2.1	<b>вариантная_часть</b>	3.7.3
базисный_символ	2.1	список_компонент	3.7
графический_символ	2.1	<b>вид_связи</b>	6.1
<b>базисное_объявление</b>	3.1	спецификация_параметров	6.1
базисный_элемент_объявления	3.9	<b>вторичный_сегмент</b>	10.1
		сегмент_компиляции	10.1
		<b>вызов_функции</b>	6.4
		префикс	4.1

<sup>1)</sup> См. сноски на с. 300.

простейшее выражение	4.4	объявление родовых параметров	12.1
<b>выражение</b>	4.4	перечислимый литерал	3.5.1
индексированная компонента	4.1.1	полное объявление типа	3.3.1
индекс входа	9.5	простое имя	4.1
квалифицированное выражение	4.7	родовая конкретизация	12.3
объявление компонент	3.7	связывание аргумента	2.8
объявление объекта	3.2	спецификация задачи	9.1
объявление родовых параметров	12.1	спецификация пакета	7.1
объявление числа	3.2	спецификация параметров цикла	5.5
оператор возврата	5.8	спецификация подпрограммы	6.1
оператор выбора	5.4	список идентификаторов	3.2
оператор присваивания	5.2	<b>имя</b>	4.1
преобразование типа	4.6	вызов функции	6.4
простейшее выражение	4.4	обозначение типа	3.3.2
родовой фактический параметр	12.3	объявление переименования	8.5
связывание аргумента	2.8	объявление родовых параметров	12.1
связывание дискриминанта	3.7.2	оператор возбуждения исключительной ситуации	11.3
связывание компонент	4.3	оператор вызова входа	9.5
спецификация дискриминантов	3.7.1	оператор вызова процедуры	6.4
спецификация параметров	6.1	оператор выхода	5.7
указатель атрибута	4.1.4	оператор перехода	5.9
условие	5.3	оператор прекращения задачи	9.10
фактический параметр	6.4	оператор присваивания	5.2
<b>вырезка</b>	4.1.2	подсегмент	10.2
имя	4.1	префикс	4.1
<b>генератор</b>	4.8	простейшее выражение	4.4
простейшее выражение	4.4	родовая конкретизация	12.3
<b>графический символ</b>	2.1	родовой фактический параметр	12.3
символьный литерал	2.5	связывание аргумента	2.8
строковый литерал	2.6	условие выбора исключительной ситуации	11.2
<b>диапазон значений</b>	3.5	фактический параметр	6.4
дискретный диапазон	3.6	фраза использования	8.4
отношение	4.4	фраза представления компоненты	13.4
уточнение диапазона значений	3.5	<b>индексированная компонента</b>	4.1.1
фраза представления компоненты	13.4	имя	4.1
<b>декларативная часть</b>	3.9	<b>индекс входа</b>	9.5
оператор блока	5.6	оператор приема	9.5
тело задачи	9.1	<b>инструкция</b>	2.8
тело пакета	7.1	<b>квалифицированное выражение</b>	4.7
тело подпрограммы	6.3	генератор	4.8
<b>десятичный литерал</b>	2.4.1	простейшее выражение	4.4
числовой литерал	2.4	<b>компиляция</b>	10.1
<b>дискретный диапазон</b>	3.6	<b>литерал с указанием системы счисления</b>	2.4.2
вырезка	4.1.2	числовой литерал	2.4
объявление входа	9.5	<b>логическая операция</b>	4.5
спецификация параметров цикла	5.5	<b>метка</b>	5.1
условие выбора	3.7.3	оператор	5.1
уточнение диапазонов индексов	3.6	множитель	4.4
<b>дискриминативная часть</b>	3.7.1	терм	4.4
незавершенное объявление типа	3.8.1	мультипликативная операции	4.5
объявление приватного типа	7.4	терм	4.4
полное объявление типа	3.3.1	<b>незавершенное объявление типа</b>	3.8.1
<b>заглушка</b>	10.2	объявление типа	3.3.1
тело	3.9	<b>обозначение</b>	6.1
<b>идентификатор</b>	2.3	родовая конкретизация	12.3
инструкция	2.8	спецификация подпрограммы	6.1
незавершенное объявление типа	3.8.1	тело подпрограммы	6.3
обозначение	6.1	<b>обозначение типа</b>	3.3.2
объявление входа	9.5	квалифицированное выражение	4.7
объявление переименования	8.5	объявление отложенной константы	7.4
объявление подтипа	3.3.2		
объявление приватного типа	7.4		

объявление переименования	8.5	таймированный вызов входа	9.7.3
объявление родовых параметров	12.1	условный вызов входа	9.7.2
оператор включения кода	13.8	<b>оператор вызова процедуры</b>	6.4
определение подтипа индекса	3.6	простой оператор	5.1
отношение	4.4	<b>оператор выхода</b>	5.7
преобразование типа	4.6	простой оператор	5.1
родовой фактический параметр	12.3	<b>оператор задержки</b>	9.6
спецификация дискриминантов	3.7.1	альтернатива с задержкой	9.7.1
спецификация параметров	6.1	простой оператор	5.1
спецификация подпрограммы	6.1	<b>оператор отбора</b>	9.7
указание подтипа	3.3.2	составной оператор	5.1
фактический параметр	6.4	<b>оператор перехода</b>	5.9
<b>обработка исключительной ситуации</b>	11.2	простой оператор	5.1
оператор блока	5.6	<b>оператор прекращения задачи</b>	9.10
тело задачи	9.1	простой оператор	5.1
тело пакета	7.1	<b>оператор приема</b>	9.5
тело подпрограммы	6.3	альтернатива приема	9.7.1
<b>объявление входа</b>	9.5	составной оператор	5.1
спецификация задачи	9.1	<b>оператор присваивания</b>	5.2
<b>объявление задачи</b>	9.1	простой оператор	5.1
базисное объявление	3.1	<b>оператор цикла</b>	5.5
последующий элемент объявления	3.9	составной оператор	5.1
<b>объявление исключительной ситуации</b>	11.1	<b>операция наивысшего старшинства</b>	4.5
базисное объявление	3.1	<b>операция отношения</b>	4.5
<b>объявление компонент</b>	3.7	отношение	4.4
список компонент	3.7	<b>описание контекста</b>	10.1.1
<b>объявление объекта</b>	3.2	сегмент компиляции	10.1
базисное объявление	3.1	<b>определение действительного типа</b>	3.5.6
<b>объявление отложенной константы</b>	7.4	определение типа	3.3.1
базисное объявление	3.1	<b>определение комбинированного типа</b>	3.7
<b>объявление пакета</b>	7.1	определение типа	3.3.1
базисное объявление	3.1	<b>определение неуточненного регулярного типа</b>	3.6
библиотечный сегмент	10.1	определение регулярного типа	3.6
последующий элемент объявления	3.9	<b>определение перечисляемого типа</b>	3.5.1
<b>объявление переименования</b>	8.5	определение типа	3.3.1
базисное объявление	3.1	<b>определение подтипа индекса</b>	3.6
<b>объявление подпрограммы</b>	6.1	определение неуточненного регулярно- го типа	3.6
базисное объявление	3.1	<b>определение подтипа компонент</b>	3.7
библиотечный сегмент	10.1	объявление компонент	3.7
последующий элемент объявления	3.9	<b>определение производного типа</b>	3.4
<b>объявление подтипа</b>	3.3.2	определение типа	3.3.1
базисное объявление	3.1	<b>определение регулярного типа</b>	3.6
<b>объявление приватного типа</b>	7.4	определение типа	3.3.1
объявление родовых параметров	12.1	родовое определение типа	12.1
объявление типа	3.3.1	<b>определение ссылочного типа</b>	3.8
<b>объявление родовых параметров</b>	12.1	определение типа	3.3.1
родовая формальная часть	12.1	родовое определение типа	12.1
<b>объявление типа</b>	3.3.1	<b>определение типа</b>	3.3.1
базисное объявление	3.1	полное объявление типа	3.3.1
<b>объявление числа</b>	3.2	<b>определение уточненного регулярного типа</b>	3.6
базисное объявление	3.1	объявление объекта	3.2
<b>оператор</b>	5.1	определение регулярного типа	3.6
последовательность операторов	5.1	<b>определение целого типа</b>	3.5.4
<b>оператор блока</b>	5.6	определение типа	3.3.1
составной оператор	5.1	<b>основание системы счисления</b>	2.4.2
<b>оператор включения кода</b>	13.8	литерал с указанием системы счисления	2.4.2
простой оператор	5.1	отношение	4.4
<b>оператор возбуждения исключительной си- туации</b>	11.3	выражение	4.4
простой оператор	5.1	<b>перечисляемый литерал</b>	3.5.1
<b>оператор возврата</b>	5.8	спецификация перечисляемого литерала	3.5.1
простой оператор	5.1	подсегмент	10.2
<b>оператор выбора</b>	5.4	вторичный сегмент	10.1
составной оператор	5.1	<b>полное объявление типа</b>	3.3.1
<b>оператор вызова входа</b>	9.5	объявление типа	3.3.1
простой оператор	5.1	<b>порядок</b>	2.4.1

десятичный литерал	2.4.1	прочий специальный символ	...
литерал с указанием системы счисления	2.4.2	графический символ	2.1
последовательность операторов	5.1	пустой оператор	5.1
альтернатива оператора выбора	5.4	простой оператор	5.1
альтернатива приема	9.7.1		
альтернатива с задержкой	9.7.1	расширенная цифра	2.4.2
обработчик исключительной ситуации	11.2	целое число с указанием системы счисления	2.4.2
оператор блока	5.6	родовая конкретизация	12.3
оператор приема	9.5	базисное объявление	3.1
оператор цикла	5.5	библиотечный сегмент	10.1
селективное ожидание	9.7.1	последующий элемент объявления	3.9
таймированный вызов входа	9.7.3	родовая спецификация	12.1
тело задачи	9.1	родовое объявление	12.1
тело пакета	7.1	родовая фактическая часть	12.3
тело подпрограммы	6.3	родовая конкретизация	12.3
условный вызов входа	9.7.2	родовая формальная часть	12.1
условный оператор	5.3	родовая спецификация	12.1
последующий элемент объявления	3.9	родовое объявление	12.1
декларативная часть	3.9	базисное объявление	3.1
преобразование типа	4.6	библиотечный сегмент	10.1
простейшее выражение	4.4	последующий элемент объявления	3.9
префикс	4.1	родовое определение типа	12.1
атрибут	4.1.4	объявление родовых параметров	12.1
вырезка	4.1.2	родовое связывание	12.3
индексированная компонента	4.1.1	родовая фактическая часть	12.3
селектируемая компонента	4.1.3	родовой фактический параметр	12.3
простейшее выражение	4.4	родовое связывание	12.3
множитель	4.4	родовой формальный параметр	12.3
простое выражение	4.4	родовое связывание	12.3
адресная фраза	13.5		
диапазон значений	3.5	связывание аргумента	2.8
оператор задержки	9.6	инструкция	2.8
отношение	4.4	связывание дискриминанта	3.7.2
указание погрешности представления плавающего типа	3.5.7	уточнение дискриминантов	3.7.2
указание погрешности представления фиксированного типа	3.5.9	связывание компоненты	4.3
условие выбора	3.7.3	агрегат	4.3
фраза выравнивания	13.4	связывание параметра	6.4
фраза длины	13.2	часть с фактическими параметрами	6.4
фраза представления компоненты	13.4	сегмент компиляции	10.1
простое имя	4.1	компиляция	10.1
адресная фраза	13.5	селективное ожидание	9.7.1
вариантная часть	3.7.3	оператор отбора	9.7
заглушка	10.2	селектируемая компонента	4.1.3
имя	4.1	имя	4.1
метка	5.1	селектор	4.1.3
оператор блока	5.6	селектируемая компонента	4.1.3
оператор приема	9.5	символьный литерал	2.5
оператор цикла	5.5	имя	4.1
родовой формальный параметр	12.3	перечисляемый литерал	3.5.1
связывание дискриминанта	3.7.2	селектор	4.1.3
селектор	4.1.3	символ операции	6.1
спецификация задачи	9.1	имя	4.1
спецификация пакета	7.1	обозначение	6.1
тело задачи	9.1	родовой формальный параметр	12.3
тело пакета	7.1	селектор	4.1.3
указатель атрибута	4.1.4	символ подчеркивания	...
условие выбора	3.7.3	идентификатор	2.3
формальный параметр	6.4	целое число	2.4.1
фраза подключения контекста	10.1.1	целое число с указанием системы счисления	2.4.2
фраза представления комбинированного типа	13.4	символ пробела	...
фраза представления перечисляемого типа	13.3	базисный графический символ	2.1
простой оператор	5.1	символ управления форматом	...
оператор	5.1	базисный символ	2.1
		соответствующее тело	3.9



подсегмент	10.2	уточнение для плавающего типа	3.5.7
тело	3.9	<b>указание погрешности представления фиксированного типа</b>	3.5.9
<b>составной оператор</b>	5.1	уточнение для фиксированного типа	3.5.9
оператор	5.1	<b>указание подтипа</b>	3.3.2
<b>специальный символ</b>	...	генератор	4.8
базисный графический символ	2.1	дискретный диапазон	3.6
<b>спецификация дискриминантов</b>	3.7.1	объявление объекта	3.2
дискриминантная часть	3.7.1	объявление подтипа	3.3.2
<b>спецификация задачи</b>	9.1	определение неуточненного регулярно-го типа	3.6
объявление задачи	9.1	определение подтипа компонент	3.7
<b>спецификация пакета</b>	7.1	определение производного типа	3.4
объявление пакета	7.1	определение ссылочного типа	3.8
родовая спецификация	12.1	определение уточненного регулярного типа	3.6
<b>спецификация параметров</b>	6.1	<b>указатель атрибута</b>	4.1.4
формальная часть	6.1	атрибут	4.1.4
<b>спецификация параметров цикла</b>	5.5	<b>унарная аддитивная операция</b>	4.5
схема итераций	5.5	простое выражение	4.4
<b>спецификация перечисляемого литерала</b>	3.5.1	<b>условие</b>	5.3
определение перечисляемого типа	3.5.1	альтернатива отбора	9.7.1
<b>спецификация подпрограммы</b>	6.1	оператор выхода	5.7
заглушка	10.2	схема итераций	5.5
объявление переименования	8.5	условный оператор	5.3
объявление подпрограммы	6.1	<b>условие выбора</b>	3.7.3
объявление родовых параметров	12.1	альтернатива оператора выбора	5.4
родовая спецификация	12.1	вариант	3.7.3
тело подпрограммы	6.3	связывание компоненты	4.3
<b>список идентификаторов</b>	3.2	<b>условие выбора исключительной ситуации</b>	11.2
объявление исключительной ситуации	11.1	обработчик исключительной ситуации	11.2
объявление компонент	3.7	<b>условный вызов входа</b>	9.7.2
объявление объекта	3.2	оператор отбора	9.7
объявление отложенной константы	7.4	<b>условный оператор</b>	5.3
объявление родовых параметров	12.1	составной оператор	5.1
объявление числа	3.2	<b>уточнение</b>	3.3.2
спецификация дискриминантов	3.7.1	указание подтипа	3.3.2
спецификация параметров	6.1	<b>уточнение диапазона значений</b>	3.5
<b>список компонент</b>	3.7	определение целого типа	3.5.4
вариант	3.7.3	уточнение	3.3.2
определение комбинированного типа	3.7	уточнение для плавающего типа	3.5.7
<b>строковый литерал</b>	2.6	уточнение для фиксированного типа	3.5.9
простейшее выражение	4.4	<b>уточнение диапазонов индексов</b>	3.6
символ операции	6.1	определение уточненного регулярного типа	3.6
<b>схема итераций</b>	5.5	уточнение	3.3.2
оператор цикла	5.5	<b>уточнение дискриминантов</b>	3.7.2
<b>таймированный вызов входа</b>	9.7.3	уточнение	3.3.2
оператор отбора	9.7	<b>уточнение для плавающего типа</b>	3.5.7
<b>тело</b>	3.9	определение действительного типа	3.5.6
последующий элемент объявления	3.9	уточнение	3.3.2
<b>тело библиотечного сегмента</b>	10.1	<b>уточнение для фиксированного типа</b>	3.5.9
вторичный сегмент	10.1	определение действительного типа	3.5.6
<b>тело задачи</b>	9.1	уточнение	3.3.2
соответствующее тело	3.9	<b>фактический параметр</b>	6.4
<b>тело пакета</b>	7.1	связывание параметра	6.4
соответствующее тело	3.9	<b>формальная часть</b>	6.1
тело библиотечного сегмента	10.1	объявление входа	9.5
<b>тело подпрограммы</b>	6.3	оператор приема	9.5
библиотечный сегмент	10.1	спецификация подпрограммы	6.1
соответствующее тело	3.9	<b>формальный параметр</b>	6.4
тело библиотечного сегмента	10.1	связывание параметра	6.4
<b>терм</b>	4.4	<b>фраза выравнивания</b>	13.4
простое выражение	4.4	фраза представления комбинированного типа	13.4
<b>терминирующая альтернатива</b>	9.7.1	<b>фраза длины</b>	13.2
альтернатива селективного ожидания	9.7.1	фраза представления типа	13.1
<b>указание погрешности представления плавающего типа</b>	3.5.7		

<b>фраза_использования</b>	8.4	<b>body</b>	...
базисный_элемент_объявления	3.9	заглушка	10.2
описание_контекста	10.1.1	тело_задачи	9.1
последующий_элемент_объявления	3.9	тело_пакета	7.1
<b>фраза_подключения_контекста</b>	10.1.1	<b>case</b>	...
описание_контекста	10.1.1	вариантная_часть	3.7.3
<b>фраза_представления</b>	13.1	оператор_выбора	7.4
базисный_элемент_объявления	3.9	<b>constant</b>	...
спецификация_задачи	9.1	объявление_объекта	3.2
<b>фраза_представления_комбинированного_типа</b>	13.4	объявление_отложенной_константы	7.4
фраза_представления_типа	13.1	объявление_числа	3.2
<b>фраза_представления_компоненты</b>	13.4	<b>declare</b>	...
фраза_представления_комбинированного_типа	13.4	оператор_блока	5.6
<b>фраза_представления_перечисленного_типа</b>	13.3	<b>delay</b>	...
фраза_представления_типа	13.1	оператор_задержки	9.6
<b>фраза_представления_типа</b>	13.1	<b>delta</b>	...
фраза_представления	13.1	родовое_определение_типа	12.1
<b>целое_число</b>	2.4.1	указание_погрешности_представления_фиксированного_типа	3.5.9
десятичный_литерал	2.4.1	<b>digits</b>	...
основание_системы_счисления	2.4.2	родовое_определение_типа	12.1
порядок	2.4.1	указание_погрешности_представления_плавающего_типа	3.5.7
<b>целое_число_в_указанной_системе_счисления</b>	2.4.2	<b>do</b>	...
литерал_с_указанием_системы_счисления	2.4.2	оператор_приема	9.5
<b>цифра</b>	...	<b>E</b>	...
базисный_графический_символ	2.1	порядок	2.4.1
буква_или_цифра	2.3	<b>else</b>	...
расширенная_цифра	2.4.2	выражение	4.4
целое_число	2.4.1	селективное_ожидание	9.7.1
<b>часть_с_фактическими_параметрами</b>	6.4	условный_вызов_входа	9.7.2
вызов_функции	6.4	условный_оператор	5.3
оператор_вызова_входа	9.5	<b>elsif</b>	...
оператор_вызова_процедуры	6.4	условный_оператор	5.3
<b>числовой_литерал</b>	2.4	<b>end</b>	...
простейшее_выражение	4.4	вариантная_часть	3.7.3
<b>abort</b>	...	оператор_блока	5.6
оператор_прекращения_задачи	9.10	оператор_выбора	5.4
<b>abs</b>	...	оператор_приема	9.5
множитель	4.4	оператор_цикла	5.5
операция_наивысшего_старшинства	4.5	определение_комбинированного_типа	3.7
<b>accept</b>	...	селективное_ожидание	9.7.1
оператор_приема	9.5	спецификация_задачи	9.1
<b>access</b>	...	спецификация_пакета	7.1
определение_ссылочного_типа	3.8	таймированный_вызов_входа	9.7.3
<b>all</b>	...	тело_задачи	9.1
селектор	4.1.3	тело_пакета	7.1
<b>and</b>	...	тело_подпрограммы	6.3
выражение	4.4	условный_вызов_входа	9.7.2
логическая_операция	4.5	условный_оператор	5.3
<b>array</b>	...	фраза_представления_комбинированного_типа	13.4
определение_неуточненного_регулярного_типа	3.6	<b>entry</b>	...
определение_уточненного_регулярного_типа	3.6	объявление_входа	9.5
<b>at</b>	...	<b>exception</b>	...
адресная_фраза	13.5	объявление_исключительной_ситуации	11.1
фраза_выравнивания	13.4	объявление_переименования	8.5
фраза_представления_компоненты	13.4	оператор_блока	5.6
<b>begin</b>	...	тело_пакета	7.1
оператор_блока	5.6	<b>exit</b>	...
тело_задачи	9.1	оператор_выхода	5.7
тело_пакета	7.1	<b>for</b>	...
тело_подпрограммы	6.3	адресная_фраза	13.5
		схема_итераций	5.5
		фраза_длины	13.2
		фраза_представления_комбинированного_типа	13.4

фраза_представления_перечисляемого_типа	13.3	<b>package</b>	...
<b>function</b>	...	заглушка	10.2
родовая_конкретизация	12.3	объявление_переименования	8.5
спецификация_подпрограммы	6.1	родовая_конкретизация	12.3
<b>generic</b>	...	спецификация_пакета	7.1
родовая_формальная_часть	12.1	тело_пакета	7.1
<b>goto</b>	...	<b>pragma</b>	...
оператор_перехода	5.9	инструкция	2.8
<b>if</b>	...	<b>private</b>	...
условный_оператор	5.3	объявление_приватного_типа	7.4
<b>in</b>	...	спецификация_пакета	7.1
вид_связи	6.1	<b>procedure</b>	...
объявление_родовых_параметров	12.1	родовая_конкретизация	12.3
отношение	4.4	спецификация_подпрограммы	6.1
спецификация_параметров_цикла	5.5	<b>raise</b>	...
<b>is</b>	...	оператор_возбуждения_исключительной_ситуации	11.3
вариантная_часть	3.7.3	<b>range</b>	...
заглушка	10.2	определение_подтипа_индекса	3.6
объявление_подтипа	3.3.2	родовое_определение_типа	12.1
объявление_приватного_типа	7.4	уточнение_диапазона_значений	3.5
объявление_родовых_параметров	12.1	фраза_представления_компоненты	13.4
оператор_выбора	5.4	<b>record</b>	...
полное_объявление_типа	3.3.1	определение_комбинированного_типа	3.7
родовая_конкретизация	12.3	фраза_представления_комбинированного_типа	13.4
спецификация_задачи	9.1	<b>rem</b>	...
спецификация_пакета	7.1	мультипликативная_операция	4.5
тело_задачи	9.1	<b>renames</b>	...
тело_пакета	7.1	объявление_переименования	8.5
тело_подпрограммы	6.3	<b>return</b>	...
<b>limited</b>	...	оператор_возврата	5.8
объявление_приватного_типа	7.4	спецификация_подпрограммы	6.1
<b>loop</b>	...	<b>reverse</b>	...
оператор_цикла	5.5	спецификация_параметров_цикла	5.5
<b>mod</b>	...	<b>select</b>	...
мультипликативная_операция	4.5	селективное_ожидание	9.7.1
фраза_выравнивания	13.4	таймированный_вызов_входа	9.7.3
<b>new</b>	...	условный_вызов_входа	9.7.2
генератор	4.8	<b>separate</b>	...
определение_производного_типа	3.4	заглушка	10.2
родовая_конкретизация	12.3	подсегмент	10.2
<b>not</b>	...	<b>subtype</b>	...
множитель	4.4	объявление_подтипа	3.3.2
операция_наивысшего_старшинства	4.5	<b>task</b>	...
отношение	4.4	заглушка	10.2
<b>null</b>	...	спецификация_задачи	9.1
список_компонент	3.7	тело_задачи	9.1
простейшее_выражение	4.4	<b>terminate</b>	...
пустой_оператор	5.1	терминирующая_альтернатива	9.7.1
<b>of</b>	...	<b>then</b>	...
определение_неуточненного_регулярного_типа	3.6	выражение	4.4
определение_уточненного_регулярного_типа	3.6	условный_оператор	5.3
<b>or</b>	...	<b>type</b>	...
выражение	4.4	незавершенное_определение_типа	3.8.1
логическая_операция	4.5	объявление_приватного_типа	7.4
селективное_ожидание	9.7.1	объявление_родовых_параметров	12.1
таймированный_вызов_входа	9.7.3	полное_объявление_типа	3.3.1
<b>others</b>	...	спецификация_задачи	9.1
условие_выбора	3.7.3	<b>use</b>	...
условие_выбора_исключительной_ситуации	11.2	адресная_фраза	13.5
<b>out</b>	...	фраза_длины	13.2
вид_связи	6.1	фраза_использования	8.4
объявление_родовых_параметров	12.1	фраза_представления_комбинированного_типа	13.4

фраза_представления_перечисляемого_типа	13.3	определение_неуточненного_регулярно-	
хот	...	го_типа	3.6
выражение	4.4	определение_перечисляемого_типа	3.5.1
логическая_операция	4.5	родовая_фактическая_часть	12.3
when	...	список_идентификаторов	3.2
альтернатива_оператора_выбора	5.4	уточнение_диапазонов_индексов	3.6
альтернатива_отбора	9.7.1	уточнение_дискриминанта	3.7.2
вариант	3.7.3	фраза_использования	8.4
обработчик_исключительной_ситуации	11.2	фраза_подключения_контекста	10.1.1
оператор_выхода	5.7	часть_с_фактическими_параметрами	6.4
while	...	—	...
схема_итераций	5.5	бинарная_аддитивная_операция	4.5
with	...	порядок	2.4.1
объявление_родовых_параметров	12.1	унарная_аддитивная_операция	4.5
фраза_подключения_контекста	10.1.1		...
"	...	десятичный_литерал	2.4.1
строковый_литерал	2.6	литерал_с_указанием_системы_счисления	2.4.2
#	...	селектируемая_компонента	4.1.3
литерал_с_указанием_системы_счисления	2.4.2	"	...
&	...	диапазон_значений	3.5
бинарная_аддитивная_операция	4.5	/	...
	...	мультипликативная_операция	4.5
атрибут	4.1.4	/=	...
квалифицированное_выражение	4.7	операция_отношения	4.5
оператор_включения_кода	13.8	:	...
символьный_литерал	2.5	объявление_исключительной_ситуации	11.1
()	...	объявление_компонент	3.7
агрегат	4.3	объявление_объекта	3.2
вырезка	4.1.2	объявление_отложной_константы	7.4
дискриминантная_часть	3.7.1	объявление_переименования	8.5
индексированная_компонента	4.1.1	объявление_родовых_параметров	12.1
инструкция	2.8	объявление_числа	3.2
квалифицированное_выражение	4.7	оператор_блока	5.6
объявление_входа	9.5	оператор_цикла	5.5
оператор_присва	9.5	спецификация_дискриминантов	3.7.1
определение_неуточненного_регулярно-		спецификация_параметров	6.1
го_типа	3.6	:=	...
определение_перечисляемого_типа	3.5.1	объявление_компоненты	3.7
подсегмент	10.2	объявление_объекта	3.2
преобразование_типа	4.6	объявление_родовых_параметров	12.1
простейшее_выражение	4.4	объявление_числа	3.2
родовая_фактическая_часть	12.3	оператор_присваивания	5.2
родовое_определение_типа	12.1	спецификация_дискриминантов	3.7.1
указатель_атрибута	4.1.4	спецификация_параметров	6.1
уточнение_диапазонов_индексов	3.6	;	...
уточнение_дискриминантов	3.7.2	адресная_фраза	13.5
фактический_параметр	6.4	вариантная_часть	3.7.3
формальная_часть	6.1	дискриминантная_часть	3.7.1
часть_с_фактическими_параметрами	6.4	инструкция	2.8
*	...	заглушка	10.2
мультипликативная_операция	4.5	незавершенное_объявление_типа	3.8.1
**	...	объявление_входа	9.5
множитель	4.4	объявление_задачи	9.1
операция_наивысшего_старшинства	4.5	объявление_исключительной_ситуации	11.1
+	...	объявление_компоненты	3.7
бинарная_аддитивная_операция	4.5	объявление_объекта	3.2
порядок	2.4.1	объявление_отложной_константы	7.4
унарная_аддитивная_операция	4.5	объявление_пакета	7.1
	...	объявление_переименования	8.5
агрегат	4.3	объявление_подпрограммы	6.1
индексирования_компонента	4.1.1	объявление_подтипа	3.3.2
инструкция	2.8	объявление_приватного_типа	7.4
оператор_прекращения_задачи	9.10	объявление_родовых_параметров	12.1
		объявление_числа	3.2
		оператор_блока	5.6
		оператор_включения_кода	13.8

оператор_возбуждения_исключительной_ситуации	11.3	<	...
оператор_возврата	5.8	операция_отношения	4.5
оператор_выбора	5.4	< <	...
оператор_вызова_входа	9.5	метка	5.1
оператор_вызова_процедуры	6.4	< =	...
оператор_выхода	5.7	операция_отношения	4.5
оператор_задержки	9.6	< >	...
оператор_перехода	5.9	объявление_родовых_параметров	12.1
оператор_прекращения_задачи	9.10	определение_подтипа_индекса	3.6
оператор_приема	9.5	родовое_определение_типа	12.1
оператор_присваивания	5.2	=	...
оператор_цикла	5.5	операция_отношения	4.5
полное_объявление_типа	3.3.1	= >	...
пустой_оператор	5.1	альтернатива_оператора_выбора	5.4
родовая_конкретизация	12.3	альтернатива_отбора	9.7.1
родовое_объявление	12.1	вариант	3.7.3
селективное_ожидание	9.7.1	обработчик_исключительной_ситуации	11.2
список_компонент	3.7	родовое_связывание	12.3
таймированный_вызов_входа	9.7.3	связывание_аргумента	2.8
тело_задачи	9.1	связывание_дискриминанта	3.7.2
тело_пакета	7.1	связывание_компоненты	4.3
тело_подпрограммы	6.3	связывание_параметра	6.4
терминирующая_альтернатива	9.7.1	>	...
условный_вызов_входа	9.7.2	операция_отношения	4.5
условный_оператор	5.3	> =	...
формальная_часть	6.1	операция_отношения	4.5
фраза_выравнивания	13.4	> >	...
фраза_длины	13.2	метка	5.1
фраза_использования	8.4		...
фраза_подключения_контекста	10.1.1	альтернатива_оператора_выбора	5.4
фраза_представления_комбинированного_типа	13.4	вариант	3.7.3
фраза_представления_компоненты	13.4	обработчик_исключительной_ситуации	11.2
фраза_представления_перечисляемого_типа	13.3	связывание_дискриминанта	3.7.2
		связывание_компоненты	4.3

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Абстрактный тип данных 187

Агрегат 57

—неоднозначный 112

—позиционный 57, 112

—с поименованными компонентами 110

Агрегат-массив 52, 53

Агрегат-структура 57

Активная задача 257

Альтернатива выбора 95

—отбора 260

Атрибут 20

—входа 265

—задачи 265

—массива 51,52

—перечисляемого типа 20, 21

—плавающего типа 37

—регулярного типа 51, 52

—фиксированного типа 39

—целого типа 20, 21

Базисное объявление 179

Базовый тип 68

Бесконечный цикл 99

Библиотека 244

Библиотечная программа 244

Библиотечный сегмент 243

Блокировка задач 258

Буферизация (для задач) 278

Ввод-вывод 204

—текстовый 235

Взаимная зависимость

—задач 258

—при рекурсивном объявлении типов 86

Взаимное исключение событий 262

Вид связи формальных параметров 129

Видимая часть пакета 181, 197

Видимость

—для задач 252

—для пакетов 197

—идентификатора 161, 197

—непосредственная 183

Владелец (задач) 253

Возбуждение исключительной ситуации 12, 29, 279

Восходящее проектирование 245

Вторичный сегмент 243

Вход задачи 251

Вызов

—входа задачи 263

----таймированный 264

----условный 263

—задачи 257

—процедуры 29, 130

---рекурсивный 150

Вызываемая задача 257

Вызывающая задача 257

Выражение 110, 113

—динамическое 27

—для абсолютной погрешности представления 38, 39

—простейшее 18, 113

—простое 113, 114

—с квалификатором 163

—статическое 27

Вырезка из массива 55

Генератор 75

Главная программа 243

Границы диапазонов индексов массива 43, 44

Двоичное дерево 153

Действительный тип 36

**Декларативная часть 11****Диапазон**

--значений 16

--дискретный 45

**Дискретный тип 15****Дискриминант 65****Заглушка 245****Задача 250**

--активная 257

--в состоянии аварийного завершения 270

--вызываемая 257

--вызывающая 257

--выполнение которой закончено 257

--выполнение которой прекращено 270

--полностью завершенная 257

**Задача-владелец 253****Закрытая альтернатива 260****Заккрытие файла 207****Зарезервированные слова Ады 13****Значение 110**

--составное 110

--пустое 110

--скалярное 110

**Идентификатор 10,13**

--(имя) цикла 99, 100

**Имена 109****Имя файла 205****Индекс 42**

--компоненты 110

--файла прямого доступа 218

**Инструкция транслятору (pragma) 165**

--определяемая реализацией 166

--определяемая языком 166

--priority 277, 278

**Исключительная ситуация 29, 279, 280**

--ввода-вывода 285, 286

--определяемая пользователем 280

--предопределенная 279, 280

**Исполняемая часть 11****Использование пакетов 187****Квалификатор типа 163****Комбинированный тип 54**

--с дискриминантами 54, 65--68

--пустой 54

--с вариантами 54, 118

**Комментарий 9****Компонента**

--массива 42

--поименованная 112

--структуры или комбинированного типа 54

**Конкретизация 199****Константа 16**

--отложенная 188

**Контекст 243, 244****Лексикографический порядок 53****Лексическая единица 10, 13****Литерал**

--действительный 14

--перечисляемый 15

--символьный 11

--числовой 11

**Локализация данных 187****Локальное объявление 102****Локальные объекты 102****Массив 42**

--динамический 44

--многомерный 42

--неуточненный 42, 51

--одномерный 42

--статический 44

--уточненный 42, 44

**Метка 97****Механизм рандеву 257, 258****Множитель 113****Модельные числа 36, 37****Набор символов**

--основной 9

--расширенный 9

--символов кода ASCII 9, 14

Начальное значение, принимаемое по умолчанию 129, 130

**Недоступный объект 78****Незавершенное объявление типа 83****Неизменяемые характеристики файла 207****Неоднозначное выражение 163**

## Неоднозначный

- агрегат 112, 163

- вызов подпрограммы 147

## Неуточненный регуляторный тип 43

- Неявное объявление 45, 100

- Нисходящее проектирование 245

- Нормализованные числа 37

- Область действия идентификатора 161, 197

- Обозначение\_типа 51, 69, 75

## Обработка

- декларативной части 102, 161

- исключительных ситуаций 279

- при параллельно протекающих процессах 294

- пакета 196

- Обработчик исключительной ситуации 280, 281

- Объединение файлов 214

- Объект 16, 78

- Объявление 15

- базисное 179

- входа 251

- дискриминантов 65, 66

- задач 250

- исключительной ситуации 279, 280

- комбинированных типов 54

- с дискриминантами 65

- констант 16

- массивов 142–151

- незавершенное 83

- неявное 45, 100

- отложенных констант 188

- пакетов 178

- переменных 16

- подпрограмм 29, 130

- частных типов 182

- регулярных типов 142–151

- родовое 200

- структур 57

- типов 15

- рекурсивное 84, 86

- Ограниченный частный тип 182

## Оператор

- блока 95, 102

- включения кода 95

- возбуждения исключительной ситуации (raise) 281

- возврата (return) 95, 136

- выбора (case) 95, 96

- вызова входа задачи 95, 252

- выхода (exit) 95, 100

- «если» (if) 11, 33, 34

- задержки (delay) 95, 259

- отбора (select) 95, 259–264

- перехода (goto) 95, 98

- прекращения задачи (abort) 95, 270, 271

- приема (accept) 95, 252

- присваивания 11

- простой 12, 95

- пустой 95, 99

- селективного ожидания 260

- составной 11, 95

- таймированного вызова входа 263, 264

- условного вызова входа 263

- цикла 12, 99

## Операция 25

- бинарная 25

- возведения в степень 25

- вычитания 26

- деления 25

- логическая 115

- сокращенная 113

- «меньше» 26

- «неравно» 26

- отношения 26, 52, 53

- проверки принадлежности 26

- «равно» 26

- сложения 26

- умножения 25

- унарная 25

- abs 25

- and 25

- in 26

- mod 25

- not 26

- not in 26

- rem 25

- Описание контекста 243, 244

- Определяемая пользователем исключительная ситуация 280

- Освобождение памяти 78

- Основание системы счисления числового литерала 14

- Основной набор символов языка Ада 9

- Открытая альтернатива 260

- Открытие файла 205

- Отложенная константа 188

- Отношение 26, 113

- Очереди (вызовов задач) 257

- Пакет 28, 178

- ASCII 14



- CALENDAR 259
- DIRECT\_IO 201, 202, 218
- ENUMERATION\_IO 22
- FIXED\_IO 41
- FLOAT\_IO 47
- INTEGER\_IO 29
- IO\_EXCEPTIONS 285
- SEQUENTIAL\_IO 210
- TEXT\_IO 200, 201, 235
- Параллельный процесс 250
- Параметр
  - входной (in) 129
  - выходной (out) 129
  - изменяемый (inout) 129
  - фактический 132
  - формальный 129
  - цикла 45
- Перекрытие
  - значений 163
  - переменных 163
  - подпрограмм 147
- Переменная 9
- Перечисляемый
  - литерал 15
  - тип 14, 15
- Плавающий действительный тип 36
- Погрешность представления
  - абсолютная 38
  - относительная 36, 37
- Подавление проверок 284
- Подпрограмма 29, 128
- Подсегмент 245
- Подтип 68
- Позиционный
  - агрегат-массив 57
  - агрегат-структура 57
- Полное завершение (задачи) 257
- Порождающий
  - сегмент 245
  - тип 71
- Порядок 37
- Последовательное выполнение операторов
- Последовательный файл 209
- Правила компиляции и перекompиляции 249
- Предопределенная исключительная ситуация 29, 279, 280
- Предопределенный тип 15
  - BOOLEAN 16
  - CHARACTER 15
  - FLOAT 36
  - INTEGER 15
  - STRING 16, 54
  - NATURAL 66
- Преобразования типов 42, 72, 163, 164
- Приватная часть пакета 181
- Приватный тип 182
- Признак конца
  - страницы 236
  - строки 236
  - файла 237
- Приоритет 277, 278
- Приостановка задачи 252
- Программа
  - ACCESS\_GRADES 88
  - ACCESS\_MAX3 77
  - ACCESS\_SHIP\_RATE 90
  - ACCR\_INTEREST 139
  - BANK\_MAINT 224
  - CHAR\_MAX3 17
  - CREATE\_12\_TRANS\_FILES 208
  - CURR\_TRANSACTION\_PROC 229
  - DATE\_CONVERSION 60
  - DAY\_CONVERSION 28
  - EXC\_DAY\_CONVERSION 282
  - EXCEP\_MERGE\_PROC\_GRADES 286
  - EXCEP\_PLANT\_SCHED 295
  - GATHER\_BANK\_STATISTICS 255
  - GR\_POINT\_AVE 120
  - HEAVY 19
  - INVENTORY\_REPORT 31
  - INVENTORY 24
  - LAST\_HIRED 58
  - MAX3 9
  - MERGE\_PROC\_GRADES 215
  - NAME\_PHONE 103
  - PAYROLL 41
  - PLANT\_SCHED 265
  - POSTING\_PROC 232
  - RACES 92
  - RECUR\_PROC\_GRADES 153
  - REPORT\_GEN 238
  - SEQ\_PROC\_GRADES 210
  - SHIP\_RATE 47
  - UP\_MONDAY 21
  - YIELD\_COMPUTATION 168
- Программный сегмент 128
- Производный тип 68, 71
- Простейшее выражение 18, 113
- Простое выражение 113, 114
- Простой оператор 12, 95
- Процедура 29, 128
- Процесс компиляции 244
- Пустой диапазон 45, 46, 99

- Разделитель 11
- Раздельная компиляция 189
- Разделяемая переменная 271, 272
- Размер файла прямого доступа 220
- Разыменование 76
- Рандеву 250
- Распространение исключительной ситуации 283, 284, 295
- Расширенный набор символов языка Ада 9
- Регулярный тип 42, 51
  - неуточненный 43, 51
  - уточненный 43, 44
- Ресентерабельный 151
- Режим обмена информацией с файлом 205
- Рекурсивное объявление типа 84, 86
- Рекурсивный вызов подпрограмм 150
- Ромбик 51
- Родовая конкретизация 199
- Родовое объявление 199
  - типа 199
- Родовой пакет 199
  - фактический параметр 199
  - формальный параметр 199
- Родовые средства 199
- Сегмент компиляции 243
- Селектор 110
- Символьный литерал 11
- Синхронизация задач 253
- Скалярный тип 42
- Согласование
  - компонент 52
  - фактических и формальных параметров 130-133
    - позиционное 133, 134
    - поименованное 133, 134
- Согласующиеся компоненты массива 52
- Создание файлов 205
- Сокращенная форма
  - дизъюнкции 113
  - конъюнкции 113
  - логических операций 113
- Соккрытие информации 182, 185
- Составной оператор 12, 95
- Состояния задач 257
- Спецификация
  - дискриминантов 67
  - задачи 251
  - обработчиков исключительных ситуаций 281
  - пакета 178
  - параметров 128
  - подпрограммы 29, 130
  - процедуры 130
- Список\_компонент 54
- Ссылочный тип 75
- Страница (при работе с пакетом TEXT\_IO) 236
- Строка (при работе с пакетом TEXT\_IO) 236
- Строки 11, 54
  - пустые 14
- Структура 54
  - с вариантами 70, 118
- Сцепление 14, 54
- Текущий индекс файла 218
- Тело
  - задачи 252
  - пакета 189
  - подпрограммы 29, 128
- Терм 113
- Терминирующая альтернатива 260
- Тип 15
  - базовый 68
  - действительный 36
  - дискретный 15
  - «задача» 251
  - комбинированный 54
  - ограниченный приватный 182
  - определяемый рекурсивно 84, 86
  - перечисляемый 15
  - плавающий 36
  - порождающий 71
  - предопределенный 15
  - приватный 68, 182
  - производный 68, 71
  - регулярный 42
  - скалярный 42
  - ссылочный 68, 75
  - фиксированный 38
  - целый 15
  - числовой 36
- Точка синхронизации 253, 271
- Указание точности (погрешности представления) 36, 68-70
- Указатель 75
- Универсальный
  - действительный тип 163
  - фиксированный тип 40

- целый тип 163
- Уничтожение объекта 78
- Уничтожение файла 208
- Условие\_выбора 95, 96
- Условный вызов входа 263
- Уточнение 15, 68, 82
  - диапазона значений 15, 36, 70
  - диапазонов индексов 44, 70, 71
  - дискриминантов 67, 70, 71
- Уточнения для ссылочного типа 82
  - для формальных параметров 134
- Уточненный регулярный тип 43

- Файл 204, 205
  - внешний 204, 205
  - внутренний 205
  - последовательный 209
  - прямого доступа 218
- Фактический параметр 132
- Фиксированный действительный тип 38
- Форма записи с селектором 110
- Формальная часть 128
  - процедуры 128
- Формальный параметр 129
- Фраза
  - использования (USE) 180
  - подключения контекста (with) 181

- представления 179, 180
- Функция 29, 128, 135–138

### Целый

- тип 15
- числовой литерал 14

### Цикл 12

- бесконечный 99
- «для» (for) 46
- поименованный 99, 100
- «пока» (while) 12

### Числовой литерал 11

- действительный 14
- целый 14
- тип 36

Член «последовательность\_операторов» 46, 257

Экспорт ресурсов 181, 187

Элемент файла 204

Язык ассемблера 95

# ОГЛАВЛЕНИЕ

<b>Предисловие редактора перевода . . . . .</b>	<b>5</b>
<b>Предисловие . . . . .</b>	<b>6</b>
<b>Глава 1. Введение . . . . .</b>	<b>9</b>
1.1. Знакомство с языком Ада . . . . .	9
1.2. Дальнейшие сведения о лексических единицах . . . . .	11
1.3. Сведения о типах и объектах . . . . .	14
1.4. Сведения о выражениях . . . . .	25
1.5. Сведения о пакетах, подпрограммах и исключительных ситуациях . . . . .	28
Упражнения . . . . .	34
<b>Глава 2. Действительные, регулярные и комбинированные типы . . . . .</b>	<b>36</b>
2.1. Действительные типы . . . . .	36
2.2. Регулярные типы . . . . .	42
2.3. Операции с регулярными типами . . . . .	52
2.4. Комбинированные типы . . . . .	56
2.5. Подтипы и производные типы . . . . .	68
Упражнения . . . . .	73
<b>Глава 3. Ссылочные типы . . . . .</b>	<b>75</b>
3.1. Введение в ссылочные типы . . . . .	75
3.2. Рекурсивные объявления ссылочных типов . . . . .	84
Упражнения . . . . .	93
<b>Глава 4. Прочие операторы Ады и комбинированные типы с вариантами . . . . .</b>	<b>95</b>
4.1. Простые и составные операторы Ады . . . . .	95
4.2. Имена, значения и выражения . . . . .	112
4.3. Комбинированные типы с вариантами . . . . .	118
Упражнения . . . . .	125
<b>Глава 5. Подпрограммы: процедуры и функции . . . . .</b>	<b>128</b>
5.1. Процедуры . . . . .	128
5.2. Функции . . . . .	135
5.3. Прикладная программа, в которой применяются функции и процедуры . . . . .	138
5.4. Перекрытие подпрограмм . . . . .	147
5.5. Рекурсивные вызовы подпрограмм . . . . .	150
Упражнения . . . . .	159
<b>Глава 6. Декларативные части и инструкции транслятору . . . . .</b>	<b>161</b>
6.1. Обработка декларативных частей . . . . .	161
6.2. Преобразования типов . . . . .	163

6.3. Инструкции транслятору . . . . .	165
6.4. Обзор основных особенностей Ады . . . . .	167
Упражнения . . . . .	176
<b>Глава 7. Пакеты . . . . .</b>	<b>178</b>
7.1. Спецификации пакетов и приватные типы . . . . .	178
7.2. Тела пакетов . . . . .	189
7.3. Правила видимости для пакетов . . . . .	197
7.4. Объявления переименования . . . . .	198
7.5. Введение в родовые пакеты . . . . .	199
Упражнения . . . . .	203
<b>Глава 8. Пакеты ввода-вывода в языке Ада . . . . .</b>	<b>204</b>
8.1. Введение в пакеты ввода-вывода . . . . .	204
8.2. Обработка последовательных файлов . . . . .	210
8.3. Обработка файлов прямого доступа . . . . .	218
8.4. Обработка файлов с помощью пакета TEXT_IO . . . . .	235
Упражнения . . . . .	242
<b>Глава 9. Структура программы и вопросы компиляции . . . . .</b>	<b>243</b>
9.1. Сегменты компиляции и процесс компиляции . . . . .	243
9.2. Подсегменты и заглушки . . . . .	245
9.3. Правила компиляции и перекомпиляции . . . . .	248
Упражнения . . . . .	249
<b>Глава 10. Задачи . . . . .</b>	<b>250</b>
10.1. Задачи и механизм рандеву . . . . .	250
10.2. Операторы и атрибуты для задач . . . . .	259
10.3. Разделяемые переменные и инструкции транслятору PRIORITY . . . . .	271
Упражнения . . . . .	278
<b>Глава 11. Исключительные ситуации . . . . .</b>	<b>279</b>
11.1. Объявление и возбуждение исключительных ситуаций . . . . .	279
11.2. Исключительные ситуации ввода-вывода. Пакет IO_EXCEPTIONS . . . . .	285
11.3. Обработка исключительных ситуаций при параллельно протекающих процессах . . . . .	294
Упражнения . . . . .	299
<b>Приложения . . . . .</b>	<b>300</b>
Приложение А. Предопределенные атрибуты языка . . . . .	300
Приложение Б. Предопределенные инструкции для транслятора языка . . . . .	307
Приложение В. Предопределенное окружение языка . . . . .	310
Приложение Г. Словарь терминов . . . . .	328
Приложение Д. Сводка синтаксиса языка Ада . . . . .	334
Ссылки . . . . .	334
<b>Предметный указатель . . . . .</b>	<b>343</b>

**Уважаемый читатель!**

**Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, изд-во «Мир».**

Учебное издание

Юджин Василеску

**ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АДА**

Заведующий редакцией д-р техн. наук А. Л. Щёрс

Зам. зав. редакцией Э. Н. Бадиков

Редактор М. Ю. Григоренко

Художник А. В. Захаров

Художественные редакторы Н. М. Иванов, О. Н. Адаскина

Технический редактор Е. С. Потапенкова

Корректор С. А. Денисова

ИБ № 6987

Сдано в набор 14.06.89. Подписано к печати 23.11.89.

Формат 70 × 100<sup>1</sup>/<sub>16</sub>. Бумага офсетная № 2. Печать офсетная.

Гарнитура таймс.

Объем 11,00 бум. л. Усл. печ. л. 28,60. Усл. кр.-отг. 57,53. Уч. изд. л. 26,99.

Изд. № 6/6477. Тираж 50 000 экз. Зак. 618. Цена 2 р. 30 к.

Издательство «Мир»

В/О «Совэкспорткнига» Государственного комитета по печати.

129820, ГСП, Москва, 1-й Рижский пер. 2.

Можайский полиграфкомбинат В/О «Совэкспорткнига»

Государственного комитета по печати.

г. Можайск, ул. Мира, 93.









UE\_DATE.SHORT\_DD ) ;

me other transaction

of the month, the 1

becomes next to last

OTHER, WRK\_LAST\_TR

TRANS\_FILE )) + 1 ;



